

SAMS

专 家 观 点

本书的内容着眼于 Linux 编程的核心特性以及复杂性的阐述：

- 强大的编程工具——包括 GCC, make, autoconf 等
- 系统编程的主题——包括进程、信号、线程、内存管理以及进程间通信的方法
- 完整描述网络编程技术
- 创建守护进程
- Linux GUI 编程——包括 X Window 系统、Qt、GNOME 以及 OpenGL
- 建立软件文档以及发布软件

[美] Kurt Wall 等著
张 辉 译

权 威 建 议

采用专家建议就能满足你在 Linux 编程上的所有需求

(第二版)

GNU/Linux

编程指南

入门·应用·精通



清华大学出版社



揭示 Linux 编程的强大功能

最终资源

本书超越了对技术的一般性讨论,向读者提供了实际的建议和深入系统的介绍。读者有了这本内容丰富的指南,就能获得广博的知识,从而发掘出 Linux 编程的全部潜力。

配套网站

www.sampublishing.com

配套网站包含本书的源代码。

权威建议

Kurt Wall 是一位 Linux 作家和顾问。他维护着 Informix 的 Linux FAQ, 还是 International Informix Users Group 中 Linux SIG(特殊兴趣小组)的主席,该小组代表了 Informix 的 Linux 用户社群。此外,他还在 Salt Lake Linux User Group 的理事会中担任技术/教育理事。Kurt 当前为 Caldera 系统工作。

详细信息

- 配置功能强大,价格低廉的 Linux 开发系统
- 深入研究类 UNIX 的输入和输出例程
- 使用信号控制程序执行
- 使用管道、消息、共享内存以及信号灯与其他程序通信
- 使用 TCP/IP, UDP 以及多播 IP 在程序中融入因特网功能
- 创建用户自己的设备驱动程序
- 编写可与流行的 K 桌面环境 (KDE) 无缝互操作的程序
- 探讨 GNU 网络对象模型环境 (GNOME) 的丰富图形界面
- 使用 OpenGL 和 Mesa 构造有趣而逼真的图形
- 使用 GNU bash shell 掌握 shell 编程
- 使用系统日志增强程序
- 学会强大的调试技术
- 使用 RPM 软件发布系统
- 轻松编写手册页
- 选择最满足需求的软件许可证

封面设计:付剑飞

读者联系电话:(010)62630320 62589259

网址:www.khp.com.cn

ISBN 7-302-05550-5



9 787302 055501 >

定价:68.00 元

SAMS

GNU/Linux 编程指南

(第二版)

入门·应用·精通

[美] Kurt Wall 等著

张 辉 译



A0996482

清华大学出版社

(京)新登字 158 号

著作权合同登记号: 01-2001-2327

内 容 提 要

本书全面而深入地介绍了 GNU/Linux 编程。首先介绍了在 Linux 上编程必备的编程工具, 然后在库函数、系统调用以及内核上阐述 Linux 编程知识, 并专门讲述了包括 TCP/IP、UDP 以及多播套接口在内的网络编程知识; 图形界面也是本书的重点内容, 本书着重讲述了文本形式的图形界面库 ncurses; 还分别讲解了真正图形化的流行系统 X Window、Qt、GNOME 以及 OpenGL 的基本编程方法; 最后, 介绍了 Bash 编程和设备驱动编程。

本书包含大量实用实例, 读者可以通过实例代码深入理解编程思想和技巧。本书另一优点是讲述了其他编程书籍通常没有提及的 RPM 包管理工具、文档编写以及发布许可证选择等内容, 这是任何准备投身于 GNU 开发工作的程序员所必须具备的知识。

本书对于所有 Linux 编程人员——无论是初学者还是高级用户——都是一本不可多得的参考资料。

Linux Programming Unleashed, Second Edition

Copyright © 2001 by Sams Publishing.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher.

本书中文简体字版由美国 Sams 公司授权清华大学出版社和北京科海培训中心出版。未经出版者书面允许不得以任何方式复制或抄袭本书内容。

版权所有, 盗版必究。

本书封面贴有清华大学出版社激光防伪标签, 无标签者不得销售。

书 名: GNU/Linux 编程指南 (第二版)
作 者: Kurt Wall, et al.
译 者: 张辉
出 版 者: 清华大学出版社 (北京清华大学校内, 邮编 100084)
印 刷 者: 北京市耀华印刷有限公司
发 行 者: 新华书店总店北京科技发行所
开 本: 787×1092 1/16 印张: 42.125 字数: 1024 千字
版 次: 2002 年 6 月第 1 版 2002 年 6 月第 1 次印刷
书 号: ISBN 7-302-05550-5/TP·3273
印 数: 0001~5000
定 价: 68.00 元

引言

很难夸大 Linux 所取得的不可思议的成功和流行。这种成功引发了对开发 Linux 应用的程序员的巨大需求，而且从长远来看，这一需求只会增加。但是，知道从哪里开始，如何开始学习为 Linux 编写程序则很困难。要对新的 Linux 程序员说的是，欢迎加入这场革命！读者将会发现本书在大多数方面是 Linux 编程的优秀指导和教材。如果读者是一位有经验的 Linux 程序员，本书也同样适用，因为它传达了一系列 Linux 编程的主题，其中也包括读者可能尚未探索过的主题。

本书的目的

本书打算向读者展示如何用 Linux 编程，如何在 Linux 上编程以及如何为 Linux 编程。它把注意力几乎全部放在了 C 语言上，因为 C 语言仍是 Linux 的“万国通用语言”。在向读者介绍一些基本开发工具之后，本书立即投入到使用 Linux I/O 模型进行编程的内容当中。本书的第 3 部分介绍了进程和同步的问题，包括线程、内存管理以及进程间通信。本书的第 5 部分专门讨论 Linux 的用户界面，它既用到了基于文本的工具，也用到了基于图形的工具（X Window 系统）。其他部分则涉及到包括 shell 编程和编写设备驱动程序在内的各种各样的话题。本书结尾的 3 章正常情况下会被一般编程的书籍所忽略：它向用户交付应用。这几章向读者展示了如何使用诸如 RPM 这样的包管理工具，以及如何创建有用的文档，还介绍了许可证的问题及其选择。读完本书，读者们就能为投身于称为“Linux”的伟大的社会性以及技术性现象做好了准备。

本书的读者

熟悉其他操作系统但是刚刚接触 Linux 的程序员会得到有关 Linux 编程的详细介绍。读者会接触到将要用到的工具以及将要开展工作的环境。

有经验的 UNIX 程序员会发现 Linux 的编程习惯用法非常熟悉。对于这类读者，本书突出了 Linux 和 UNIX 专有版本的不同之处。最大的可移植性将会是一个重要的话题，因为 Linux 运行在一种尚在变化的平台上：Intel i386, Sun Sparc 和 UltraSparc, Digital Alpha, MIPS 处理器, Power PC 以及基于 Motorola 68000 的 Macintosh 计算机。

中等水平的 C 程序员也能从本书获益。总体而言，在 Linux 编程类似于在其他类 UNIX 系统上编程，所以读者将会很快入门，成为高效的 UNIX 程序员并且理解 Linux/UNIX 深入的特质。

目 录

第 1 部分 Linux 编程工具包

第 1 章 Linux 及 Linux 编程综述.....1

1.1 Linux 变得成熟了.....1	1
1.1.1 Linux 的昨天.....1	1
1.1.2 Linux 的今天.....3	3
1.1.3 Linux 的明天.....3	3
1.2 为何选择 Linux 编程.....3	3
1.3 每章内容介绍.....5	5
1.3.1 Linux 编程工具包.....5	5
1.3.2 输入、输出、文件和目录.....5	5
1.3.3 进程和同步.....6	6
1.3.4 网络编程.....6	6
1.3.5 用户界面编程.....7	7
1.3.6 特殊编程技术.....8	8
1.3.7 补充内容.....8	8
1.4 小结.....8	8

第 2 章 设置开发系统.....9

2.1 一般性考虑.....9	9
2.2 主板和 CPU.....10	10
2.2.1 板上 I/O.....12	12
2.2.2 处理器.....12	12
2.2.3 BIOS.....13	13
2.2.4 内存.....13	13
2.2.5 机箱和电源.....13	13
2.3 用户交互硬件：视频、声音、键盘 及鼠标.....14	14
2.3.1 显卡.....14	14
2.3.2 显示器.....15	15
2.3.3 声卡.....16	16
2.3.4 键盘及鼠标.....16	16
2.4 通信设备、端口及总线.....17	17

2.4.1 调制解调器.....17	17
2.4.2 网络接口卡.....18	18
2.4.3 SCSI.....18	18
2.4.4 USB 和火线 (IEEE1394).....18	18
2.4.5 串行卡.....18	18
2.4.6 IRDA.....19	19
2.4.7 PCMCIA 卡.....19	19
2.4.8 ISA 即插即用设备.....19	19
2.5 存储设备.....19	19
2.5.1 硬盘.....20	20
2.5.2 可移动磁盘设备.....20	20
2.5.3 CD-ROM/DVD.....20	20
2.5.4 磁带备份设备.....20	20
2.6 外围设备.....21	21
2.6.1 打印机.....21	21
2.6.2 扫描仪.....21	21
2.6.3 数字相机.....22	22
2.6.4 家居自动控制设备.....22	22
2.7 完备型系统.....22	22
2.8 便携系统.....22	22
2.9 开发工具软件.....23	23
2.9.1 关键库和头文件.....23	23
2.9.2 调试器.....23	23
2.9.3 编程工具.....23	23
2.9.4 文本编辑器.....24	24
2.10 小结.....24	24

第 3 章 使用 GNU CC.....25

3.1 GNU CC 特性.....25	25
3.2 教学示例.....26	26
3.3 常用命令行选项.....29	29

3.3.1 函数库和包含文件.....	30	5.3.5 类型定义测试.....	64
3.3.2 警告和出错消息选项.....	31	5.3.6 编译器行为测试.....	65
3.4 优化选项.....	36	5.3.7 系统服务测试.....	65
3.5 调试选项.....	38	5.3.8 UNIX 变体测试.....	66
3.6 特定体系结构的选项.....	40	5.4 普通宏.....	66
3.7 GNU C 扩展.....	41	5.5 一个带注释的 autoconf 脚本.....	68
3.7.1 关于可移植性.....	41	5.6 小结.....	74
3.7.2 GNU 扩展.....	42	第 6 章 比较和合并源代码文件.....	75
3.8 gcc: 奔腾处理器的编译器.....	45	6.1 使用 diff 命令比较文件.....	75
3.9 小结.....	45	6.2 理解 diff3 命令.....	83
第 4 章 使用 GNU make 管理项目.....	46	6.3 准备源代码补丁.....	86
4.1 为何使用 make.....	46	6.3.1 patch 的命令行选项.....	86
4.2 编写 makefile.....	46	6.3.2 创建补丁.....	87
4.3 编写 makefile 的规则.....	47	6.3.3 应用补丁.....	88
4.3.1 伪目标.....	49	6.4 小结.....	88
4.3.2 变量.....	50	第 7 章 使用 RCS 和 CVS 控制版本... ..	89
4.3.3 隐式规则.....	52	7.1 基本术语.....	89
4.3.4 模式规则.....	53	7.2 使用修订控制系统 (RCS).....	90
4.3.5 注释.....	53	7.2.1 RCS 基本用法.....	90
4.4 命令行选项和参数.....	53	7.2.2 找出 RCS 文件间的不同.....	95
4.5 调试 make.....	54	7.2.3 其他 RCS 命令.....	98
4.6 常见的 make 出错信息.....	54	7.3 使用并发版本系统 (CVS).....	100
4.7 有用的 makefile 目标.....	54	7.3.1 同 RCS 相比的优点.....	100
4.8 小结.....	55	7.3.2 设置 CVS.....	100
第 5 章 创建可移植的自配置软件.....	56	7.3.3 检出源代码文件.....	102
5.1 考虑可移植性.....	56	7.3.4 将改动合并进源代码库.....	103
5.1.1 什么是程序的可移植性.....	56	7.3.5 检查改动.....	103
5.1.2 移植性的线索和技巧.....	57	7.3.6 添加和删除文件.....	104
5.2 理解 autoconf.....	58	7.3.7 解决文件冲突.....	105
5.2.1 创建 configure.in.....	58	7.3.8 CVS 命令.....	106
5.2.2 构造文件.....	58	7.3.9 CVS 选项.....	106
5.2.3 有用的 autoconf 工具.....	59	7.4 小结.....	107
5.3 内置宏.....	62	第 8 章 调试.....	108
5.3.1 候选程序测试.....	62	8.1 为使用 GDB 进行编译.....	108
5.3.2 库函数测试.....	63	8.2 使用基本的 GDB 命令.....	109
5.3.3 头文件测试.....	64	8.2.1 启动 GDB.....	109
5.3.4 结构测试.....	64		

8.2.2 在调试器中查看代码.....	111
8.2.3 检查数据	111
8.2.4 设置断点	113
8.2.5 检查并更改运行中的代码.....	114
8.3 高级 GDB 概念和命令	116
8.3.1 变量的作用域和上下文.....	116
8.3.2 遍历函数堆栈.....	117
8.3.3 操纵源代码文件.....	119
8.3.4 与 Shell 进行通信	119
8.3.5 附加到某个运行中的程序.....	119
8.4 小结	121
第 9 章 出错处理	122
9.1 出错处理简述	122
9.2 出错处理选项	122
9.3 C 语言机制.....	123
9.3.1 assert 宏	123
9.3.2 使用预编译.....	125
9.3.3 标准库函数.....	127
9.4 使用系统日志	133
9.4.1 系统日志选项.....	134

9.4.2 系统日志函数.....	135
9.4.3 用户程序.....	138
9.5 小结	139

第 10 章 使用库 140

10.1 使用编程库.....	140
10.1.1 库兼容性.....	140
10.1.2 命名和编号约定	141
10.1.3 经常使用的库.....	142
10.2 库操作工具.....	143
10.2.1 理解 nm 命令	143
10.2.2 理解 ar 命令.....	144
10.2.3 理解 ldd 命令.....	144
10.2.4 理解 ldconfig	145
10.2.5 环境变量和配置文件	145
10.3 编写并使用静态库.....	146
10.4 编写并使用共享库.....	151
10.5 使用动态加载的共享对象.....	152
10.5.1 理解 dl 接口.....	152
10.5.2 使用 dl 接口.....	154
10.6 小结.....	155

第 2 部分 输入、输出、文件和目录

第 11 章 输入和输出..... 156

11.1 基本特点和概念.....	156
11.2 理解文件描述符.....	160
11.2.1 文件描述符的概念.....	160
11.2.2 文件描述符的优缺点.....	161
11.3 使用文件描述符.....	162
11.3.1 打开关闭文件描述符.....	162
11.3.2 读写文件描述符.....	164
11.3.3 使用 ftruncate 缩短文件.....	166
11.3.4 使用 lseek 定位文件指针.....	166
11.3.5 使用 fsync 同步到硬盘	167
11.3.6 使用 fstat 获得文件信息.....	167

11.3.7 使用 fchown 改变文件所有权	172
11.3.8 使用 fchmod 改变文件读写权	172
11.3.9 使用 flock 和 fcntl 给文件上锁	172
11.3.10 使用 dup 和 dup2 调用	177
11.3.11 使用 select 同时读写多个文件	179
11.3.12 使用 ioctl.....	182
11.4 小结.....	182

第 12 章 文件和目录操作..... 183

12.1 标准文件函数.....	183
------------------	-----

12.1.1 打开和关闭文件.....	183	12.2.7 删除和改名.....	189
12.1.2 读写文件.....	184	12.2.8 使用临时文件.....	190
12.1.3 获得文件状态.....	185	12.3 目录操作.....	191
12.2 输入输出调用.....	186	12.3.1 找到当前目录.....	191
12.2.1 格式化输出.....	186	12.3.2 改变目录.....	191
12.2.2 格式化输入.....	187	12.3.3 创建和删除目录.....	191
12.2.3 字符输入输出.....	187	12.3.4 获得目录列表.....	192
12.2.4 行输入输出.....	188	12.4 特殊的 ext2 文件系统属性.....	192
12.2.5 文件定位.....	189	12.5 小结.....	195
12.2.6 缓冲区控制.....	189		

第 3 部分 进程和同步

第 13 章 进程控制.....	196	第 14 章 线程概述.....	229
13.1 Linux 进程模型.....	196	14.1 什么是线程.....	229
13.2 进程属性.....	196	14.2 __clone 函数调用.....	229
13.2.1 进程标识号.....	197	14.3 pthread 接口.....	230
13.2.2 Real 和 Effective 标识号.....	198	14.3.1 pthread 是什么.....	230
13.2.3 SetUID 和 SetGID 程序.....	198	14.3.2 何时使用 Pthread.....	231
13.2.4 用户和用户组信息.....	200	14.3.3 pthread_create 函数.....	231
13.2.5 附加的进程信息.....	201	14.3.4 pthread_exit 函数.....	231
13.3 创建进程.....	208	14.3.5 pthread_join 函数.....	232
13.3.1 使用 system 函数.....	208	14.3.6 pthread_atfork 函数.....	234
13.3.2 fork 系统调用.....	209	14.3.7 取消线程.....	234
13.3.3 exec 函数族.....	211	14.3.8 pthread cleanup 宏.....	237
13.3.4 使用 popen 函数.....	213	14.3.9 Pthread 条件.....	238
13.4 控制进程.....	214	14.3.10 pthread_equal 函数.....	238
13.4.1 等待进程——wait 函数族.....	214	14.3.11 线程属性.....	239
13.4.2 杀死程序.....	216	14.3.12 互斥.....	240
13.5 信号.....	218	14.4 小结.....	244
13.5.1 什么是信号.....	218	第 15 章 访问系统信息.....	245
13.5.2 发送信号.....	219	15.1 进程信息.....	246
13.5.3 捕获信号.....	221	15.1.1 cmdline 文件.....	246
13.5.4 检测信号.....	226	15.1.2 environ 文件.....	246
13.6 进程调度.....	227	15.1.3 fd 目录.....	247
13.7 小结.....	228	15.1.4 mem 文件.....	247

15.1.5	stat.....	247	16.1.2	calloc 函数的使用	257
15.1.6	status 文件	249	16.1.3	realloc 函数的使用	257
15.1.7	cwd 符号链接	249	16.1.4	free 函数的使用	257
15.1.8	exe 符号链接	249	16.1.5	alloca 函数的使用	258
15.1.9	maps 文件	249	16.2	内存映像文件	258
15.1.10	root 符号链接	249	16.2.1	mmap 函数的使用	259
15.1.11	statm 文件	249	16.2.2	munmap 函数的使用	260
15.2	一般系统信息	249	16.2.3	msync 函数的使用	260
15.2.1	/proc/cmdline 文件	249	16.2.4	mprotect 函数的使用	260
15.2.2	/proc/cpuinfo 文件	250	16.2.5	锁定内存	261
15.2.3	/proc/devices 文件	250	16.2.6	mremap 函数的使用	261
15.2.4	/proc/dma 文件	250	16.2.7	用内存映像实现 cat 命令	261
15.2.5	/proc/file systems 文件	250	16.3	发现并修改内存问题	263
15.2.6	/proc/interrupts 文件	250	16.3.1	一个有问题的程序	263
15.2.7	/proc/ioports 文件	250	16.3.2	Electric Fence	265
15.2.8	/proc/kcore 文件	250	16.4	小结	267
15.2.9	/proc/kmsg 文件	250	第 17 章 进程间通信	268	
15.2.10	/proc/ksyms 文件	251	17.1	管道	268
15.2.11	/proc/loadavg 文件	251	17.1.1	打开和关闭管道	269
15.2.12	/proc/locks 文件	251	17.1.2	读写管道	271
15.2.13	/proc/mdstat 文件	251	17.1.3	更简单的方法	274
15.2.14	/proc/meminfo 文件	251	17.2	FIFO	276
15.2.15	/proc/misc 文件	251	17.2.1	理解 FIFO	276
15.2.16	/proc/modules 文件	251	17.2.2	创建 FIFO	278
15.2.17	/proc/mounts 文件	251	17.2.3	打开和关闭 FIFO	279
15.2.18	/proc/pci 文件	251	17.2.4	读写 FIFO	279
15.2.19	/proc/rtc 文件	252	17.3	System V IPC 概述	282
15.2.20	/proc/stat 文件	252	17.3.1	System V IPC 的主要概念	282
15.2.21	/proc/uptime 文件	252	17.3.2	System V IPC 的问题	284
15.2.22	/proc/version 文件	252	17.3.3	Linux 和 System V IPC	285
15.2.23	/proc/net 子目录	252	17.4	共享内存	285
15.2.24	/proc/scsi 子目录	253	17.4.1	创建共享内存区	286
15.2.25	/proc/sys 子目录	253	17.4.2	附加共享内存区	288
15.3	未来内核中/proc 的变化	255	17.5	消息队列	291
15.4	小结	255	17.5.1	创建和打开消息队列	291
第 16 章 内存管理	256		17.5.2	向队列中写入消息	293
16.1	C 内存管理回顾	256	17.5.3	读取队列中的消息	295
16.1.1	malloc 函数的使用	256	17.5.4	删除消息队列	297

17.6 信号灯	299
17.6.1 创建信号灯	299
17.6.2 控制和删除信号灯	301
17.7 小结	303
第 18 章 守护进程	304
18.1 理解守护进程	304
18.2 创建守护进程	304

18.2.1 函数调用	305
18.2.2 出错处理	308
18.3 和守护进程通信	311
18.3.1 读取配置文件	311
18.3.2 向守护进程加入信号处理功能	314
18.4 小结	319

第 4 部分 网络编程

第 19 章 TCP/IP 和套接口编程	320
19.1 套接口的定义	320
19.2 通信域	321
19.3 套接口编程基础	321
19.3.1 分配套接口和初始化	321
19.3.2 完成连接的系统调用	323
19.3.3 传送数据	324
19.3.4 关闭	325
19.4 使用套接口的客户机/服务器例子	325
19.4.1 服务器的例子程序	326
19.4.2 客户机的例子程序	328
19.4.3 运行客户机和服务器的例子	331
19.4.4 使用 Web 浏览器作为客户机	331
运行服务器的例子程序	331
19.5 一个简单的 Web 服务器和 Web 客	331
户机的例子程序	331
19.5.1 实现一个简单的 Web 服务器	332
19.5.2 实现一个简单的 Web 客户机	336
19.5.3 测试 Web 服务器和 Web 客户机	338
19.5.4 使用 Netscape Navigator 作为客户	339
机运行简单的 Web 服务器	339

19.6 通过其他编程语言使用套接口	339
19.7 UNIX 域套接口的 Perl 编程	339
19.8 监视套接口活动的工具	341
19.9 小结	342

第 20 章 UDP: 用户数据报协议

20.1 UDP 概述	343
20.1.1 UDP 和 TCP 的对比	343
20.1.2 TCP 的优缺点	343
20.1.3 UDP 的优缺点	344
20.1.4 选择使用哪一种协议	344
20.2 实现一个基于 UDP 的应用	345
20.2.1 使用 UDP 发送数据	345
20.2.2 接收 UDP 数据	348
20.2.3 最少的出错检查	350
20.2.4 非阻塞 I/O	355
20.3 小结	361

第 21 章 多播套接口和非阻塞 I/O

21.1 配置 Linux 支持多播 IP	362
21.2 为支持多播 IP 重新编译 Linux 内核	363
21.3 多播 IP 广播的示例程序	363
21.3.1 使用多播 IP 广播数据	364
21.3.2 创建客户程序监听多播 IP 广播	366

21.3.3 运行多播 IP 示例程序	370
---------------------------	-----

21.4 小结	371
---------------	-----

第 5 部分 用户界面编程

第 22 章 底层终端控制

22.1 终端接口	372
22.2 控制终端	373
22.2.1 属性控制函数	374
22.2.2 速度控制函数	375
22.2.3 行控制函数	376
22.2.4 进程控制函数	377
22.3 使用终端接口	378
22.4 改变终端模式	380
22.5 使用 terminfo	382
22.5.1 terminfo 能力	382
22.5.2 terminfo 编程	383
22.5.3 发挥 terminfo 能力	387
22.6 小结	390

第 23 章 ncurses 入门

23.1 ncurses 简史	391
23.2 使用 ncurses 编译程序	392
23.3 调试 ncurses 程序	392
23.4 关于窗口	392
23.4.1 ncurses 窗口设计	393
23.4.2 ncurses 函数命名规则	394
23.5 初始化和终止	395
23.5.1 ncurses 初始化结构	395
23.5.2 ncurses 终止	395
23.5.3 说明 ncurses 初始化和终止	396
23.6 输入和输出	398
23.6.1 输出例程	398
23.6.2 输入例程	406
23.7 色彩例程	409
23.8 窗口管理	411
23.9 其他各种工具函数	412

23.10 小结	415
----------------	-----

第 24 章 ncurses 高级编程

24.1 其他 ncurses 功能	416
24.1.1 鼠标支持	416
24.1.2 菜单支持	416
24.1.3 窗体支持	416
24.2 和鼠标交互	417
24.2.1 鼠标 API 概述	417
24.2.2 鼠标控制例程	418
24.2.3 示例程序	419
24.3 使用菜单	422
24.3.1 菜单 API 概述	422
24.3.2 菜单控制例程	423
24.3.3 示例程序	428
24.4 ncurses 窗体	430
24.4.1 窗体 API 概述	430
24.4.2 窗体管理例程	431
24.4.3 示例程序	440
24.5 小结	441

第 25 章 X Windows 编程

25.1 X 的概念	443
25.2 Xlib API	444
25.2.1 XOpenDisplay	445
25.2.2 XCreateSimpleWindow 和 XCreateWindow	445
25.2.3 映射窗口和撤销映射窗口	446
25.2.4 撤销窗口	447
25.2.5 事件处理	447
25.2.6 初始化图形设备上下文和字体	448
25.2.7 在 X 窗口中绘图	449

25.2.8 一个 Xlib 的示例程序	449	26.5.4 Button 类	491
25.3 X Toolkit API	455	26.5.5 Text 类	492
25.3.1 X Toolkit 使用入门	455	26.6 小结	494
25.3.2 使用 X 工具包设置窗口部件参数	456	第 27 章 使用 GTK+ 进行 GUI 编程 ...	495
25.4 XFree86	458	27.1 GTK+ 简介	496
25.4.1 DPMS——显示器电源管理信令	458	27.1.1 在 GTK+ 中处理事件	497
25.4.2 DRI——直接显示接口	458	27.1.2 使用 GTK+ 的简短示例程序	498
25.4.3 DGA——直接图形体系结构 ..	458	27.1.3 各种 GTK 窗口部件	500
25.4.4 XV——X 视频	459	27.1.4 GTK 容器窗口部件	501
25.5 小结	459	27.2 一个用于显示 XML 文件的 GTK+ 程序	502
第 26 章 Athena、Motif 和 LessTif 窗口部件	460	27.2.1 XML 简介	502
26.1 使用 Athena 的窗口部件	460	27.2.2 James Clark 的 XML 分析器 expat	503
26.1.1 Athena 的标签窗口部件	460	27.2.3 实现 GTK+ 的 XML 显示程序	504
26.1.2 Athena 的命令按钮窗口部件 ..	461	27.2.4 运行 GTK+ 的 XML 显示程序	510
26.1.3 Athena 的列表窗口部件	464	27.3 一个使用 Notebook 窗口部件的 GUI 程序	510
26.1.4 Athena 的文本窗口部件	465	27.3.1 Notebook 窗口部件示例程序的实现	510
26.1.5 Athena 的简单菜单窗口部件 ..	468	27.3.2 实现 Drawing 窗口部件	512
26.2 使用 Motif 的窗口部件	470	27.3.3 运行 GTK Notebook 窗口部件的示例程序	515
26.2.1 Motif 的标签窗口部件	471	27.4 通过其他编程语言使用 GTK+	515
26.2.2 Motif 的列表窗口部件	472	27.4.1 通过 C++ 使用 GTK+	516
26.2.3 Motif 的文本窗口部件	474	27.4.2 通过 Perl 使用 GTK+	517
26.3 编写一个定制的 Athena 窗口部件	477	27.4.3 通过 Python 使用 GTK+	518
26.3.1 使用 fetch_url.c 文件	477	27.5 GTK+ 的 RAD 工具	518
26.3.2 使用 URL.h 文件	479	27.6 小结	519
26.3.3 使用 URLP.h 文件	480	第 28 章 使用 Qt 进行 GUI 编程	520
26.3.4 使用 URL.c 文件	481	28.1 通过重载 QWidget 类方法处理事件	521
26.3.5 测试 URLWidget	484	28.1.1 QWidget 类概述	521
26.4 在 C++ 程序中使用 Athena 和 Motif	485	28.1.2 实现 DrawWidget 类	523
26.5 使用封装 Athena 窗口部件的一个 C++ 类库	486	28.1.3 测试 DrawWidget	525
26.5.1 Component 类	487		
26.5.2 PaneWindow 类	488		
26.5.3 Label 类	490		

28.2 使用 Qt 槽和信号处理事件	526	29.3.3 使用 GLUT 创建简单的 3D 对象	541
28.2.1 派生 StateLCDWidget 类	526	29.3.4 使用 x-y-z 坐标在 3D 空间中放置对象	542
28.2.2 使用信号和槽	529	29.3.5 沿着 x-、y-、z-中任一坐标轴或所有坐标轴旋转对象	543
28.2.3 运行信号/槽示例程序	531	29.3.6 启用 Material 属性	544
28.3 用 Qt 实现 XMLview 的程序	531	29.3.7 启用深度测试	545
28.3.1 SAX2: 一个用于 XML 的简单 API	532	29.3.8 处理键盘事件	545
28.3.2 DOM: 文档目标对象	533	29.3.9 为获得动画效果更新 OpenGL 图形	545
28.4 小结	537	29.3.10 Orbits 程序清单	546
第 29 章 使用 OpenGL 和 Mesa 进行 3D 图形编程	538	29.4 纹理映像	548
29.1 需要为本章准备什么	538	29.4.1 用纹理面产生立方体	548
29.2 使用 OpenGL	539	29.4.2 创建纹理映像	549
29.3 3D 图形编程	539	29.4.3 立方体程序清单	550
29.3.1 orbits.c	539	29.5 小结	554
29.3.2 为 OpenGL 图形创建窗口并初始化 OpenGL	540		

第 6 部分 特殊编程技术

第 30 章 使用 GNU Bash 进行 Shell 编程	555	30.5.3 不确定性循环: while 和 until	569
30.1 为何使用 bash	555	30.5.4 选择结构: case 和 select	569
30.2 bash 基础知识	555	30.6 shell 函数	572
30.2.1 通配符	556	30.7 输入与输出	573
30.2.2 花括号展开式	556	30.7.1 I/O 重定向	573
30.2.3 特殊字符	557	30.7.2 字符串 I/O	574
30.3 使用 bash 变量	558	30.8 命令行处理	576
30.4 使用 bash 操作符	561	30.9 进程和作业控制	578
30.4.1 字符串操作符	561	30.9.1 Shell 的信号处理	578
30.4.2 模式匹配操作符	562	30.9.2 使用 trap	578
30.5 流控制	564	30.10 小结	580
30.5.1 条件执行: if	564	第 31 章 设备驱动程序	581
30.5.2 确定性循环: for	568	31.1 驱动程序的类型	581

31.1.1 静态链接的内核设备驱动程序	581	31.5.1 低层端口的 I/O.....	589
31.1.2 可加载的内核模块.....	582	31.5.2 使用 DMA 访问内存.....	591
31.1.3 共享库	582	31.5.3 引发使用设备驱动程序的中断	591
31.1.4 无特权用户模式程序.....	582	31.5.4 设备驱动程序分层	592
31.1.5 特权用户模式程序.....	583	31.6 简单的用户模式测试驱动程序	593
31.1.6 守护进程	583	31.7 创建内核驱动程序	594
31.1.7 字符设备与块设备的对比.....	583	31.7.1 查看源代码.....	594
31.2 怎样构造硬件.....	583	31.7.2 编译驱动程序.....	619
31.2.1 理解步进电机的工作原理.....	584	31.7.3 使用内核驱动程序.....	620
31.2.2 标准的或双向的并口.....	586	31.7.4 未来发展方向.....	621
31.3 建立开发环境.....	588	31.8 其他信息资源.....	621
31.4 调试内核级驱动程序.....	589	31.9 小结.....	622
31.5 设备驱动程序内幕.....	589		

第 7 部分 补充内容

第 32 章 软件包管理623

32.1 理解 tar 文件	623
32.1.1 创建 tar 文件	624
32.1.2 更新 tar 文件	625
32.1.3 列出 tar 文件的内容.....	626
32.1.4 从一个存档文件解出文件.....	627
32.2 理解 install 命令.....	628
32.3 理解 Red Hat 包管理器 (RPM) ...	630
32.3.1 RPM 是什么	630
32.3.2 最小要求	631
32.3.3 配置 RPM	631
32.3.4 控制构造过程: 使用 spec 文件	633
32.3.5 分析一个 spec 文件.....	634
32.3.6 构造软件包.....	637
32.4 文件层次结构标准.....	637
32.5 小结	639

第 33 章 建档..... 640

33.1 编写手册页面.....	640
33.1.1 手册页面的组成.....	640
33.1.2 手册页面的例子.....	641
33.1.3 使用 groff 命令	643
33.1.4 Linux 约定	644
33.2 使用 DocBook	645
33.2.1 DocBook 是什么.....	646
33.2.2 DocBook 标记	646
33.2.3 DocBook 文档示例.....	647
33.2.4 生成输出.....	653
33.3 小结.....	653

第 34 章 许可证的发放 654

34.1 介绍和弃权.....	654
34.2 MIT/X 风格的许可证.....	654
34.3 BSD 风格的许可证	655

34.4	Artistic 的许可证.....	656	34.5.2	GNU 库通用公共许可证 (LGPL)	
34.5	GNU 通用公共许可证	656		658
34.5.1	GNU 通用公共许可证 (GPL)		34.6	开发源代码的定义.....	658
	657	34.7	小结.....	660

第1部分 Linux 编程工具包

第1章 Linux 及 Linux 编程综述

Linux 不再是爱好者的玩具了。它已经成为几乎每一种计算基础设施，尤其是因特网必不可少的组成部分。如果说编写本书第一版的 1998 年是 Linux 最终出现在美国的雷达屏幕上的一年，那么到了 1999 年 Linux 已经是一名美国正式公民了。再向前看，在 2000 年里，Linux 肯定会把自己稳固树立为因特网计算基础设施的一个基本组成部分以及新经济的一名正式成员。

1.1 Linux 变得成熟了

熟悉 Linux 的程序员尤其是应用程序开发人员都知道 Linux 市场正在爆炸性地增长。本节简要说明了这一情况发生的原因。如果你熟悉 Linux 的历史，可以直接跳到下一节。

1.1.1 Linux 的昨天

Linux 的发布历史始于 1991 年 8 月发表在 Usenet 新闻组 comp.os.minix 上的如下一篇文章，作者是一名芬兰的大学生：

```
Hello everybody out there using minix-  
I'm doing a (free) operating system (just a hobby, won't be big and  
professional like gnu) for 386 (486) AT clones. This has been brewing  
since april, and is starting to get ready. I'd like any feedback on  
things people like/dislike in minix, as my OS resembles it somewhat  
(same physical layout of the file-system (due to practical reasons)  
among other things).
```

这个学生就是 Linus Torvalds，而他所编写的“业余爱好”Linux 已经发展成为现在的 Linux。Linux 1.0 版内核发布于 1994 年 3 月 14 日。1996 年 6 月发布了 2.0 版，在 1.0 版和 2.0 版内核之间，大量的开发工作极大地提高了基本的内核功能、增加了设备支持，最重要的是扩大了可以使用的应用程序的范围。到了发布 2.0 版内核的时候，几乎没有用 Linux 执行不了的任务，只是界面尚不理想（理想的意思是“类似于 Windows”）。

注意：实际上，Linus 最初关于 Linux 的文章出现在 1991 年 7 月 3 日，但当时并没有特别提到 Linux。Linus 本人讲述的 Linux 早期开发的一段有趣的历史可以从网上得到：<http://www.li.org/li/linuxhistory.shtml>。

目前稳定的 2.2 版内核正式发布于 1999 年 1 月 25 日。编写本书的时候, 2.3 版的开发版内核处于代码冻结状态。此时内核不再增加新特性, 按照发布 2.4 版内核的目标, 代码编写工作转向排错并稳定内核的基础代码。该版本的目标是包括一个日志型文件系统、使对称多处理机子系统有更好的粒度, 以及对 POSIX 标准更完备的支持。上面的几点介绍没有涵盖增强和排错工作的所有方面。

1998 年 3 月, 当 Netscape 承诺在 GNU 计划的 GPL (General Public License, 通用公共许可证) 的一个修订版本的基础上公开 Netscape Communicator Internet 套件的源代码时, Linux 一下子进入了大众的视野。同年 7 月, 世界上最大的两家关系数据库厂商 Informix 和 Oracle 宣布把他们的数据库产品移植到了 Linux 上。1998 年 8 月, Intel 和 Netscape 公司购买了 Linux 发布商中的领头羊 Red Hat 公司的少量股票, 专项资金投入了 Linux 世界。同时, IBM 开始了它的旗舰数据库产品 DB/2 在 Linux 上移植版本的 beta 测试版。这一年中还有许多类似的开发工作在继续实施。

1999 年 Linux 成为主流 IT 市场中的年青一分子。主要的软硬件厂商继续向 Linux 公司进行投资, 随后很快又投入了风险资金。不出所料, 随着 Red Hat 和 VA Linux 首先成为上市的 Linux 公司, 从大型投资商注入的资金取得了成功的股票销售行情。

2000 年的头几个月, IBM 宣布将在它的服务器、台式机以及便携机产品线上预装 Linux。实际上, 截止到 2000 年 4 月, 大多数主要的硬件厂商都把 Linux 作为其所售计算机系统上的一种候选软件。公众对于 Linux 的兴趣与日俱增, 计算机业内几乎每周都有与 Linux 相关的消息发布。更令人兴奋的是大众媒体对于 Linux 的关注。实际上, Linux 已经不再仅仅是爱好者们的一个玩具了。

与此同时, 那个位于雷蒙德^①市的小^②软件公司微软再也坐不住了。在微软著名(也可以说臭名昭著)的万圣节文件泄漏之后, 迫使该公司对这种迅速发迹的操作系统予以回应。万圣节文档是微软内部的备忘录, 该文档详细地分析了微软对于 Linux 对其市场霸权尤其是服务器操作系统 Windows NT 的威胁, 并且讨论了用以对付 Linux 挑战的策略。根据万圣节文档制订的策略, 微软发起了一个“恐惧、不可靠、怀疑”(FUD, 三个字母分别代表 Fear、Uncertainty 和 Doubt) 运动, 其后果是很快否认 Mindcraft 公司有关服务器性能的评测报告, 并发表违背常识的带有偏见的白皮书“Linux 神话”。Linux 社群中的大多数人都嘲笑微软的这些伎俩, 而在更大范围的计算机世界里, 特别是鉴于微软最初的结论曾经还是开放源代码的开发方式要比传统的软件开发模型更优越这一事实, 通常很少有人会严肃看待微软关于 Linux 或开放源代码软件所发表的声明。然而, 微软被迫用 FUD 的策略来对付 Linux 的事实表明它对于自己的产品没有信心。

注意: Mindcraft 公司最初的研究报告载于 <http://www.mindcraft.com/whitepapers/nts4rhlinux.html>。“Linux 神话”一文可通过 <http://www.microsoft.com/NTServer/nts/news/msnw/LinuxMyths.asp> 在线阅览。

① 译者注: 雷蒙德(Redmond)是位于美国华盛顿州西雅图东北的一个小城, 微软总部所在地。

② 译者注: 众所周知, 微软是世界上最大的软件公司, 作者在这里用“小”软件公司来形容它, 一方面表示作者对微软的鄙视, 另一方面也隐含了微软公司名称中的“小”(Micro)。

1.1.2 Linux 的今天

作为一种服务器级的操作系统，Linux 已经成熟了。提供 Web 服务器的 Linux 系统遍布全球，而且越来越多的商业用户使用 Linux 系统提供文件和打印服务。它既被当作邮件服务器的一种候选平台，也被当作一种强壮而安全的防火墙。业界认为 Linux 在因特网服务器市场将会取得更大份额（绝大多数是原 Windows NT 和 Windows 2000 的市场份额），并且预计 Linux 还会在嵌入式设备市场、因特网设备市场和专门性服务器市场占据主导地位。

Linux 的企业级特性，比如支持多处理器、支持大型文件系统、日志型文件系统以及密集型计算和高可用性集群技术，也会逐步成熟。2.2 版内核最多支持 16 个 CPU，比 2.0 版最多支持 4 个 CPU 有了提高。密集型计算集群技术让 Linux 用户可以创建包含数十乃至数百台廉价的日用型个人计算机的系统，从而达到超级计算机水平的处理速度，而价格同 Cray、SGI 或 SUN 的超级计算机相比极为低廉。高可用性集群技术让 Linux 系统在一个或多个要害部件（比如电源或硬盘）出现故障时，仍然能够继续正常工作。

桌面上的 Linux 也在继续完善。KDE 桌面提供的图形用户界面（GUI，Graphical User Interface）在易用性和可配置方面都能和微软的 Windows 相媲美。但与 Windows 不同的是，KDE 只是运行在健壮而灵活的操作系统之上为改善视觉效果而加入的一层软件。KDE 强大的命令行界面是用户垂手可得的强大工具。Linux 有不少于四种以上办公软件套件：Applixware、StarOffice 以及作为 KDE 计划一部分的 KOffice 都已投入实际使用。Corel 发行了它的 Linux 办公套件 WordPerfect Office 2000，以及它自己的 Linux 发布版本 Corel LINUX OS。在大量 Linux 应用程序和工具软件的基础上，与微软 Office 类似的办公软件的出现使得 Linux 成为 Windows 在桌面系统上的一个竞争对手。

1.1.3 Linux 的明天

Linux 将会何去何从？Linux 因其强壮、稳定、功能强大的特点而在服务器领域极为流行，但它因为在实用性方面持续遭遇挑战而仍然没有在桌面上超越 Windows。在渐露头角的因特网设备，比如防火墙和路由器市场上，Linux 是充满希望的，但还没有在数量上得到完全验证。缩微版本的 Linux 在嵌入式设备，比如电话和机器控制器市场上逐渐流行起来。随着 Linux 厂商，比如 Caldera、Red Hat、VA Linux 以及 Cobalt 作为上市公司逐步成熟，它们必然会推进自己的商业计划和策略，并指导 Linux 的未来。

1.2 为何选择 Linux 编程

人们为什么会用 Linux 编程，又为什么会为 Linux 编程呢？这个问题的答案可能会和从事 Linux 编程的人数一样多。但我认为，大多数答案可以归结为以下几类。

首先，Linux 编程很有意思——这就是我做这件事的原因。其次，Linux 是自由软件。第三，Linux 是开放的。它没有隐藏的接口，也没有未公开的功能或 API（Application Programming Interface，应用程序编程接口）——在了解操作系统的功能方面，没有人或组

织会不公正地享有什么特权。

第四，如果你不喜欢什么东西的工作方式，你可以直接拿到源代码并修改它，这意味着你的公司或工作不会受制于其他公司的计划。不幸的是，一些人，特别是信息系统经理，把软件维护看作是缺点，因为软件维护不但增加了项目成本，而且给已经超负荷的信息系统员工增加了过多的任务。这种态度源于一种误解——拥有源代码并不强迫你一定要用它。流行的应用软件，比如 Apache Web 服务器，是处于主动维护和开发状态之下的，所以如果有需要，那么产品的核心开发小组完全可能提供所需的更新和增强特性。关键在于使用自由软件这样做很简单，因为如果愿意你可以得到源代码。

最后，这也是我认为最重要的原因。Linux 程序员属于一个特殊的群体。从一定层次上看，每个人都需要归属于某种精神，并产生对这种精神的认同感。对于 Windows 程序员而言是这样，对于 Linux 程序员而言也是这样，对于去参加教堂、俱乐部和运动队的人们而言也是这样。从另一个更深的层次上看，是否能进入这个群体取决于一个人的能力、技术和才干，而不取决于财富、相貌或人际关系。例如，Linus Torvalds 很少因为某些看似合理的观点而改动内核，而只有正在实际运行的代码才能说服他（“给我看代码”是他的口头禅）。

我不认为 Linux 代表着一种精英文化。一个人在群体中的作用取决于他能够满足需要的程度，而无论他的工作是编写代码、整理文档还是帮助初学者。做这些事情既需要有一定的技术和才干，也需要有做这些事情的愿望。当你加入进来成为 Linux 编程群体的一员之后，你会发现做这件事情不但有趣，而且有意义。我就是这样认为的。分析到最后，Linux 是一个知识共享的群体。

为什么要读这本书？随着 Linux 本身以及 Linux 产业的不断发展变化，对于 Linux 程序员的需求也在增长。无论是初学编程的新手还是富有经验的程序员，刚刚接触 Linux 时都会对其中大量的工具软件和技术知识望而生畏，难以决定从何处下手。本书就是专门为这样的读者编写的。它向读者介绍了 Linux 编程常用的工具软件和技术。我真诚地希望本书的内容能够帮助读者在 Linux 实用编程方面打下坚实的基础。我深信，在你读完本书后就已为进一步深入研究 Linux 做好了准备。

注意：严格地说，Linux 不是 UNIX。UNIX 是一个注册商标，需要满足一大串条款并且支付可观的费用才能被许可使用它。Linux 只是 Unix 的克隆，在运行特性上与 Unix 相似而已。Linux 所有的内核代码都由 Linus Torvalds 以及其他几位核心黑客手工编写的。许多在 Linux 上运行的程序也都是手工编写的，但是，也有大量的软件只是简单的从其他操作系统尤其是 UNIX 和类 UNIX 操作系统上移植而来。

更重要的是，Linux 是一个符合 POSIX 规范的操作系统。POSIX 是由电子和电气工程师协会（Institute of Electrical and Electronic Engineers, IEEE）提出的一系列标准，用于定义一个可移植的操作系统接口。实际上，Linux 为什么与 UNIX 这么相像，原因就在于 Linux 遵循 POSIX 标准。

1.3 每章内容介绍

本节概述了本书的内容。虽然本书是按照便于读者从头到尾阅读的方式编写的，但是如果你是一位有经验的 Linux 程序员，你就可以直接翻到特定的一章，找出完成手头工作所需的信息。

1.3.1 Linux 编程工具包

本书的第 1 章介绍了 Linux 的编程环境，阐明了可以使用的工具软件及其用法。这一章是把 Linux 编程放在正在发生的 Linux 现象的大环境中来进行介绍的。

第 2 章讨论了创建开发系统，比如选择硬件、平衡性能价格比，以及在创建理想的开发环境时涉及到的问题。可一旦你建起了一个系统，但不知道如何使用编译器 gcc 还是无法进行 Linux 编程。第 3 章教你如何使用 gcc，内容包括它的调用语法、它的命令行选项和参数以及 GNU 对 ANSI C 标准的扩展。

在第 4 章你会碰到 make 程序。任何稍大些的程序都是由多个源代码文件构成的，这些源代码文件之间有着复杂的依赖关系。第 4 章阐明了如何使用 make 程序来简化这类编程项目的管理工作。

第 5 章介绍了 autoconf 程序，它是 Linux 程序员必不可少的工具。工具程序 autoconf 可以让你编写能够在多种不同平台上编译和执行的源代码。它能够配置源代码以便可以在任何给定的 CPU 和操作系统的组合环境中正确编译的过程得以自动完成。

把对源代码的变动融入编程项目是第 6 章的主题。合并代码是开发过程的一部分，比如对错误代码的修改和补丁，这些都是由后来人而不是程序的最初开发者提交的。diff 和 patch 分别是用于比较文件和把变动合并到项目里的主要工具程序。

教会你跟踪源代码的修改是第 7 章的焦点。RCS (Revision Control System, 修订控制系统) 和 CVS (Concurrent Versions System, 并发版本系统) 能够控制对源代码的访问、跟踪对源代码的修改，还可以管理版本编号和代码发布过程。这对于由多个程序员共同完成的项目来说尤其重要，因为它可以防止一个程序员不小心破坏其他程序员的工作。

在第 8 章中你会遇到 GNU 的调试器 gdb。编程过程中不可避免地会出现错误。这一章教你如何使用 GNU 的调试器找出并修正这些错误。它还再次用到了编译器 gcc 的特性，这些特性是对调试过程的补充。

类似地，周全地处理运行期的错误也是一项必不可少的编程任务。第 9 章介绍了出错处理。一般程序都应该在出错时尽量减小影响。ANSI C 标准和 Linux 内核都具有检测并响应异常出错状态的能力。本章教你可以使用哪些出错处理工具及其用法。

第 10 章讨论编程库的使用，以此结束本书的第一部分。除了其他作用以外最重要的是，编程库通过把频繁使用的多个例程集中到一处以利于在多个项目和程序中使用相同的代码，方便地实现了代码重用。

1.3.2 输入、输出、文件和目录

编写本书第 2 部分的目的有三个。第一，它简要回顾了几乎在每个 Linux 程序中都会

用到的基本 C 编程技术。第二,更为重要的是,它向读者介绍了标准的 Linux 编程惯用语,并且阐明了何时、为何以及怎样使用它们。最后,本章讨论了构成 Linux 编程基础的一些基本概念。

不需要输入或不产生输出的程序是非常少见的。第 11 章回顾了标准 C 语言的 I/O 例程,并且展示了如何使用传统的 Linux 和 UNIX I/O 工具。作为第 11 章内容的继续和扩展,第 12 章深入探讨了 Linux 文件的抽象概念。Linux 中的任何东西都可以按照文件的方式进行访问,这是一个关键的 Linux 概念。和前面几章类似,这一章也回顾了标准 C 语言的文件控制例程,并且教给你如何使用这些函数实现所提及的有关 Linux 文件的抽象概念。

1.3.3 进程和同步

第 3 部分介绍了 Linux 的进程模型,这是 Linux 编程的另一个核心内容。这部分内容包括进程模型本身、进程间通信以及进程怎样得到并管理内核分配给它们的资源。

粗略地说,一个进程就是一个运行中的程序。进程模型是 Linux 编程的另一个核心概念。在程序怎样运行、何时运行以及在什么环境下运行几方面, Linux 都赋予了程序员比 Windows 更多的控制权。第 13 章解释了 Linux 进程模型以及如何使用与其相关的编程工具。第 14 章通过介绍线程编程拓展了对进程的讨论。多线程编程并不是一剂包治百病的灵丹妙药,但你会看到在许多情况下它确能带来巨大的好处。这一章还介绍了一些特殊的编程要求,你会从中了解它们。

第 15 章向读者展示了如何去访问系统信息。Linux 内核能向应用软件程序员提供大量信息,包括应用软件所运行的系统的信息、该系统的状态信息以及程序可以使用的工具和服务信息。读者将学会如何取得并使用这些信息。

大多数程序员都需要以多种方式分配系统内存,即使这样做的原因几乎总是为了让程序在运行时有更大的灵活性以及避免主观上的限制。第 16 章说明了 ANSI C 和 Linux 内核为管理内存提供的功能。

读者将从第 17 章开始使用进程间通信 (Interprocess Communication, IPC),这一术语用于描述正在同一机器上运行的程序和进程间所进行的通信。IPC 是让多个进程共享信息和资源的机制。

第 18 章结束了有关进程的教学内容,在这一章中讨论了一种特殊的进程——守护进程。守护进程是在后台运行、典型情况下是等候其他程序或进程请求某种服务的进程。虽然守护进程并不难编写,但它们确有不同于普通用户级程序的特殊要求。

1.3.4 网络编程

网络可以想像成是多台独立的机器之间的 IPC (进程间通信),而不是同一机器上多个独立的程序之间的 IPC。Linux 内建了完整的网络功能。实际上,想到 Linux 时很难不想到网络。第 4 部分向读者介绍网络编程。这个主题范围很大,所以这里的内容只是介绍性的。然而在这一章学到的知识能够让你编写出功能强大,实用性强的网络程序。

第 19 章深入介绍网络编程,这一章开发的高级程序示例展示了如何在网络上进行 TCP/IP 和套接口编程。

第20章介绍了用户数据报协议 (User Datagram Protocol, UDP), 并且告诉读者如何编写使用 UDP 的程序。UDP 协议比 TCP 协议速度快, 因为它既不用担心分组丢失也不保证消息传送到目的地。因此, 对于某些应用而言它是一种理想的网络编程协议。

第21章介绍多播套接口和非阻塞套接口 I/O。多播套接口用来创建诸如聊天和视频会议之类的分布式应用程序。非阻塞套接口 I/O 是指在套接口处不等待 (或者称为阻塞) 输入到来的套接口 I/O。不等待输入 (或输出) 是需要实时响应的程序, 比如游戏。另外, 这一章还向读者展示了怎样构造一个用于套接口编程的 C++ 类库, 使得编写采用套接口的程序更容易。

1.3.5 用户界面编程

显然, 和用户交互才可能完成计算功能。Linux 允许程序在几个不同级别上和用户 (以及设备) 进行交互。在最低级别上, 每一次击键事件都能被截获, 由程序决定如何解释击键动作。目前用户交互的最高级别是通过 X Window 系统实现的。在这两个极端之间, ncurses 库提供了一种用于控制台的图形界面, 也就是说文本和字符模式的界面。第5部分涵盖了怎样使用所有这些方法编写用户界面的内容。

术语终端一词的历史可以追溯到 UNIX 的早期时代, 那时用户和系统的交互通过键盘和单行打印机 (称为电传打印机, 简称 TTY) 来进行。随着时间的推移, TTY 模型演变为一种抽象概念, 用户和设备通信的每一种形式, 比如打印机和调制解调器都被包含到这种模型里。这种模型甚至能够让你控制文本怎样在一个终端窗口中进行显示。第22章介绍了 TTY 模型。

第23章介绍 ncurses。ncurses (新版 curses) 是原始的 GUI, 它为字符模式的终端提供了复杂的屏幕控制功能。本书有两章内容介绍如何编写文本模式 GUI, 这是其中的一章。第24章继续教授 ncurses。读者将学习如何使用 ncurses 的菜单功能、同鼠标交互的功能和窗体功能。

大多数新出现的 GUI 程序都使用了 X Window 系统。第25章向读者介绍了 X Window 系统背后的客户机/服务器的原理和概念, 这一章还展示了如何使用最早的 X 编程库, Xlib 和 Xt。

读者将在第26章碰到 X 窗口部件 (X widget)。窗口部件是用于描述诸如复选框、按钮、窗口、滚动条以及对话框等的术语。Athena 和 Motif 是两种最流行和最知名的窗口部件库, 它们极大地减少了 X 程序的代码量。

第27章转入介绍 Linux 上常见的 GUI 编程工具包, 这一章向读者介绍如何使用 GTK+。GTK+ 代表 Gimp Toolkit, 它是新一代窗口部件库之一。它最初是为了供 GIMP 使用而开发的, 后来成了一种流行的 X (和 Linux) 编程工具包。它是桌面环境 GNOME 的基础库。Qt 是另一种流行的 X 编程工具包, 它是另一种流行的桌面环境 KDE 的基础库。读者会在第28章里学习 Qt。

由于像 Quake 系列的游戏大为流行, 人们对于三维编程的兴趣高涨。第29章概要介绍了使用 Mesa 的三维编程技术, Mesa 是三维图形毫无疑问的事实标准——OpenGL 3D 图形开发包的开放源代码实现。

1.3.6 特殊编程技术

坦率地说，第6部分的内容和本书的其他部分并不匹配。在这一部分中，你将学习使用 `bash` 和设备驱动程序进行 `shell` 编程。

`bash` 是 Linux 的“标准”`shell`。它也是一种完善的编程语言。第30章全面介绍了这种编程语言，该语言被选择用来进行系统初始化、关机和系统管理。如果你打算在 Linux 使用上花些时间，那么有必要熟悉 `bash` 编程。

设备驱动程序是让内核（以及用户）和各种硬件设备交互的代码模块。第31章阐述了如何编写设备驱动程序。

1.3.7 补充内容

本书最后一部分的内容通常是 Linux 编程类书籍涉及不到的：为发布而打包软件、编写文档以及选择合适的软件许可证。然而，理解如何为世界准备好你的软件、如何为它编写文档以及为它选择许可证是非常重要的。

第32章讨论包管理，包管理是指为了发布软件而制作二进制或源代码形式的完整软件包。本章介绍了传统工具 `tar`、逐渐流行起来（并且有点误称）的 `RPM`（Red Hat Package Management，Red Hat 包管理）和 `Debian` 的包管理系统。

第33章的建档是指建立用户文档，即程序的用法说明。良好的文档是一个完整的软件产品必不可少的部分，但这一点常常被忽略又经常被误导。

第34章的内容和软件许可证有关，它和文档类似，通常对这方面的介绍也不多。在开放源代码的世界里，程序员可以在多种软件许可证中进行自由地选择。本章介绍了多种许可证，强调了它们的优缺点并说明如何使用它们。这一章并没有推荐一种许可证比另一种要好，但是如果你所关注的地方本书没有介绍，那么最好找一位不错的律师，听听他的意见。

1.4 小 结

本章概述了 Linux 的历史，简要介绍了目前 Linux 以及 Linux 编程的发展状态，并对 Linux 的未来给出了一些合理的预言。随后，还讲述了 Linux 和 UNIX 的关系，并对为什么要用 Linux 编程做了一个简要的带有哲学意味的回答。最后指出笔者是多么希望你能够通过阅读本书学到许多东西。

第2章 设置开发系统

在理想世界里，每个开发人员都拥有至少一个专门用于软件开发的系统。这样一种安排最主要的好处在于，如果你在测试自己编写的新设备驱动程序时不小心搞坏了开发系统，那么你还可以继续用自己的主系统。本章指导你在有时显得混乱的一系列选择中做出正确判断，并且讨论了在为开发系统选择软硬件时需考虑的问题。

2.1 一般性考虑

本章内容不可避免地存在一些主观色彩。开发人员对于硬件系统的选择在很大程度上取决于个人的需要和偏好。阅读本章时最好同时阅读 Linux 建档计划 (Linux Documentation Project, LDP) 维护的 Linux 硬件兼容性列表 HOWTO 一文。HOWTO 最新版本的在线地址是：<http://www.linuxdoc.org/HOWTO/Hardware-HOWTO.html>。

注意： 如果你不能访问因特网，那么在大多数 Linux 系统的 `/usr/doc/HOWTO` 路径下也能找到 Linux 硬件兼容性列表 HOWTO 的电子版。

硬件兼容性列表 HOWTO 的目的有两个。首先，它列举了一系列 Linux 支持或不支持的特殊设备，这可以简化创建开发系统的任务。其次，HOWTO 还给出了其他文档，这些文档列举出了 Linux 支持或不支持的更多硬件。在多数情况下，本章并不列举所支持的特殊设备，因为这个列表不但很长而且很快就会过时。本章只是偶尔提及某些已知在 Linux 下工作良好的品牌名称或设备类型。

提示： 在购买和安装 Linux 系统之前能够访问因特网，虽说决非必要，但还是建议你能有这样的前提条件。虽然 Linux 的安装和配置比以前简单，但能够很容易地访问保存在 Web 站点上有关 Linux 的信息还是会非常方便的。

如果你发现自己在读完本章之后还需要更多的信息，特别是关于 Linux 支持的硬件信息的话，那么 Linux 在线 (Linux Online, 地址为 <http://www.linux.org>) 维护了一个“Projects”网页，网页上包含有指向许多主要开发项目的链接，其中也有指向多种硬件设备支持项目的链接。

本章集中介绍能够满足工作需要的硬件并说明如何令其发挥作用，而不是讲述怎样构建一个功能过剩的系统。为什么呢？一些读者可能由于经济限制而不能购买最新型的 CPU、大量内存、高速硬盘和 48 英寸的大显示器。其他一些读者可能愿意拥有两台以上的廉价系统而不是单独一台昂贵的机器。笔者的情况是，我发现在我的 SuperMondoMegaBodaciousModel II 计算机上开发软件能够掩盖代码效率低的诸多症状，比如极端消耗内存、离奇的 CPU 高占用率以及硬盘占用过大等（作者以诙谐的口吻说明了他

的计算机还是能够满足开发需要的，虽然开发阶段的代码效率不高，浪费了 CPU、内存和硬盘资源。))。

本章开始时建议读者应该有两套系统，这样做的原因很多，其中一些理由如下：

- 需要有一个单独的路由器/防火墙，随着越来越多的人使用 DSL 或专线访问因特网，因而把他们的系统长期暴露给了坏小子们，使得这一需求愈发重要；
- 需要有一个专门的“崩溃和烧毁”系统；
- 需要有一个为了测试目的而能够启动多种操作系统或一个操作系统不同版本的系统；
- 如果准备发布软件包，需要一个独立的、干净的系统测试安装程序或软件包（RPM 或 Debian）；
- 需要一个干净的系统用于测试软件；
- 如果不希望把自己的系统共享出去，需要为客人或家庭其他成员准备一个独立的系统；
- 需要至少一台 Linux 机器作为服务器，一天 24 小时不间断运行。

数百万行 Linux 代码中的大多数可能都是在还要比当前销售的经济型系统慢得多的系统上开发的。但是，如果有需要和足够的经济能力而购买了比建议的配置性能更高的系统，那么就会有更强的能力。本章中的建议是面向中低档开发工作站的。当然这个建议标准可以根据实际需要上下浮动。

一些基本的开发工作，比如使用本书述及的那些工具和技术，实际上都不需要高速的 CPU；如果说需要什么的话，那么大多数的软件开发工作都是 I/O 密集型而不是 CPU 或内存密集型的（编译内核的工作是上述情况的一个特例）。但是另外一些在开发过程中用到的应用软件，或者正在开发的应用程序可能对 CPU 和内存有额外的要求。比如说，编译 C++ 程序，特别是大型程序会消耗大量的计算能力。类似地，多媒体应用程序需要的计算能力通常要比开发人员在普通的编辑、编译、调试周期内需要的计算能力大。商用办公软件套件一般要求大量的内存。调试代码的要素之一，即连续单步执行监视变量的变化，肯定会消耗大量的 CPU 周期。

有的人建议说要选择一个能够在今后两三年内满足需要的系统。这不是一个明智的想法。今后一年内，你所需要的计算能力及特性的价格必然会大大下降。结果是，现在只购买自己需要的，等到明年再买那时需要的，这样做可能会更划算。另一种更经济的方案是随着时间推移不断升级现有系统的组件而不是购买一个全新的系统。如果你选用升级的方式，要特别注意不要购买太专有的部件，否则将来会限制你选用满足需要的部件的能力。

2.2 主板和 CPU

系统主板和 CPU 是任何计算机的核心部分。主板最重要的特性之一是它的物理结构，即大小、形状以及关键特性的位置。许多计算机制造商，尤其是一些主要品牌，都使用专用的主板，而这正是我们所要避免的。因修理或升级的原因而要替换主板时，专用主板只

有有限的选择（或者说是没有选择），而且代价昂贵。一些厂商为了降低制造成本而使用专用主板（因为可以减少串口、并口和其他 I/O 口的连线），而另一些则有险恶的用心。

老的 AT（或 baby AT）结构的主板具有可换性，但其上只有少量的印刷电路靠近于安置接口的机箱后端，机箱后端也只留出容纳键盘孔和鼠标的插孔。

新的 ATX 标准和 AT 标准相比有许多优点。虽然新的 ATX 主板的尺寸与 Baby AT 的近似（都与 8.5 英寸×11 英寸的纸张差不多大）。ATX 主板倒转了长短边的位置，使得长边正对机箱后端。ATX 主板的另一个优点是 ATX 机箱后端有一个标准的用金属片封装的长方形预留区域，其上有足够的预留位置来匹配特定主板上的接口，足以用堆叠的方式容纳以下接口：

- 2 个串口
- 1 个并口
- 键盘口
- 鼠标口
- 2 个 USB 口
- VGA 口
- 音频接口

同时，ATX 设计移动了 CPU 和内存芯片的位置，使其不再与整长的 I/O 卡互相妨碍，但是有一些厂商仍然设置了一些可能会产生冲突的内部接口。关于 ATX 主板的详细信息，参见 <http://www.teleport.com/~atx/>。图 2.1 和图 2.2 给出了 AT 和 ATX 主板在物理形状上的差异。

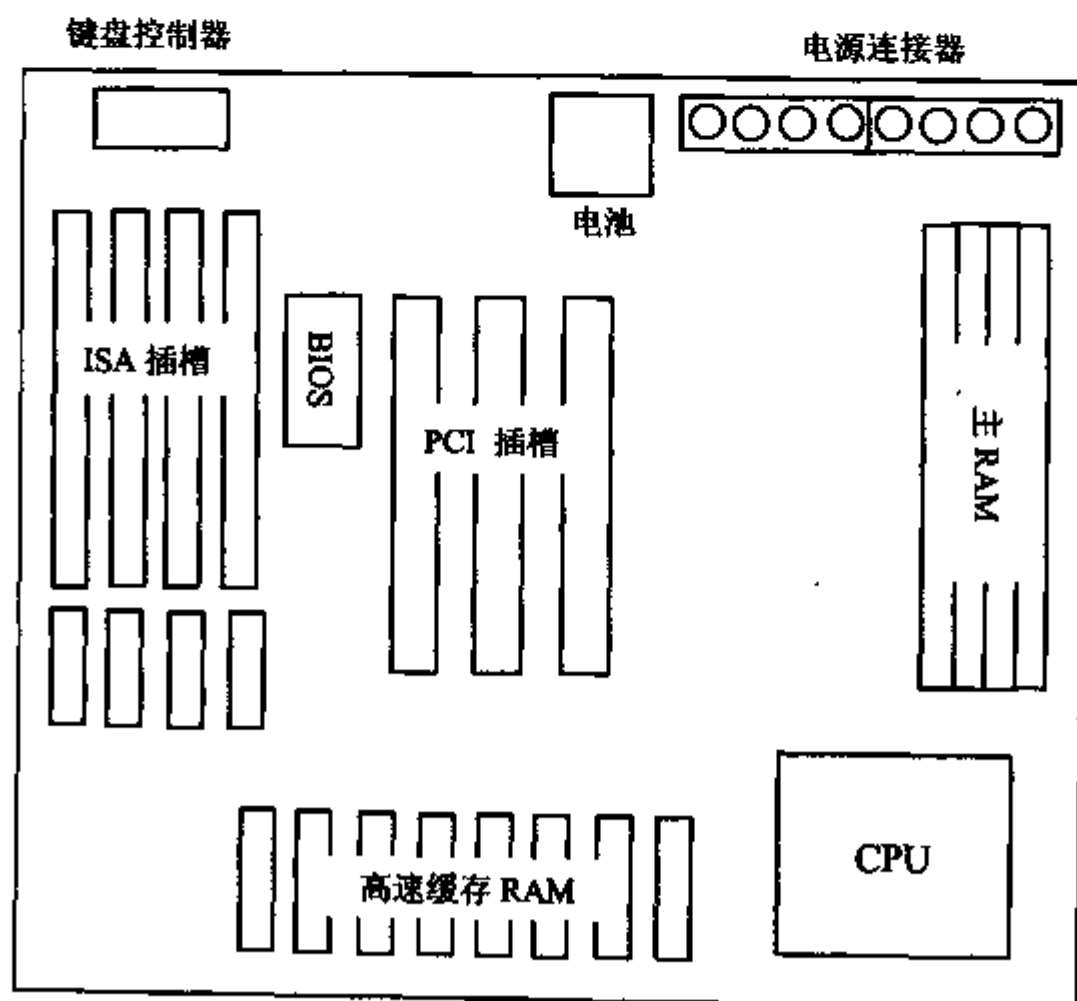


图 2.1 AT 主板布局

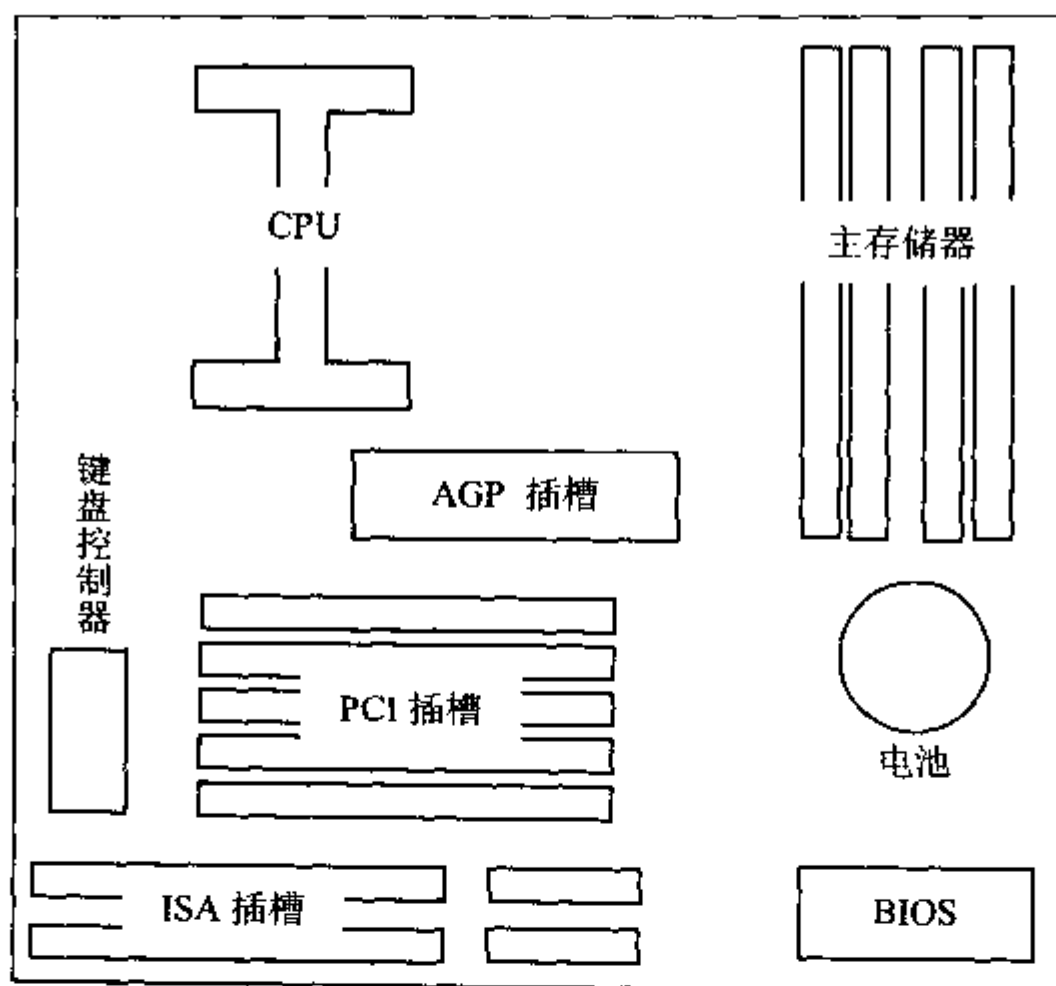


图 2.2 ATX 主板布局

上面两幅图显示了两种主板的主要部件在布局上的差异。最重要的变化是 ATX 主板将扩展插槽旋转移动了 90° ，使它们和主板的长边垂直。这样做增加了一块主板可以支持的扩展插槽的数量。

2.2.1 板上 I/O

一个典型的 Pentium 或更高级的主板在板上有 2 个串口，1 个并口，1 个键盘口，1 个鼠标口、2 个 IDE 接口和软驱接口；所有这些都能在 Linux 下运行良好。而主板上一些额外的接口，如 USB，SCSI，Ethernet，Audio 和 Video 等，可能需要做兼容性检查。

2.2.2 处理器

本节假设读者使用的是 Intel CPU 或与之兼容的处理器，比如 AMD、Cyrix 处理器。采用这样的硬件构成的廉价系统仍然能够运行大量的软件。当然，在处理器上还可以有其他诸如 Alpha、SPARC、UltraSPARC、MIPS 以及 PowerPC 体系结构的选择。如果对系统所支持的其他处理器结构感兴趣，可以在 <http://www.linux.org> 上找到有关资料。

Cyrix 和 AMD 生产与 Pentium 兼容的处理器，这些处理器有一些兼容性问题，但现在已经解决。而编译器 gcc 现在已经为 AMD 芯片进行了优化。许多人倾向于使用 Socket 7 架构的主板，这类主板能够对 Intel、Cyrix 和 AMD 的芯片都提供很好的支持。Pentium II，Pentium III，Xeon 和 Celeron 芯片也应被简单地当作 Pentium 兼容芯片。

如果非常在乎价格，那么使用 Cyrix MediaGX 处理器的系统则非常便宜。MediaGX 系统把 CPU、处理器缓存、显卡、声卡、主板芯片组以及 I/O 端口集成在两块芯片上。这种安排的一个缺点是不能用其他品牌的处理器替换 MediaGX。第二个缺点是视频系统要使用

系统的内存进行视频操作。这种设计减少了应用程序可以使用的系统内存，而且屏幕的刷新还占用了处理器到内存间的带宽。最后导致系统性能比根据处理器速度得出的期望性能低三分之一。

MediaGX 明显的优点是它们的廉价，而且从软件的角度来看所有的 MediaGX 系统都是一样的。所以，只要一台 MediaGX 系统能工作，则其他所有的系统都能工作。SuSE 公司提供了 MediaGX 的视频支持。在 http://www.suse.de/XSuSE/XSuSE_E.html 上有关于 MediaGX 视频驱动程序的更多信息。

在过去的两年里，笔者主要用于开发的机器带有一个 Pentium II 266 MHz 的处理器。虽然目前高端处理器的速度已经是这款处理器的三倍，但我还是最后才考虑升级处理器。如果我给系统增加了内存，或是用 SCSI 子系统取代 IDE 子系统，那么系统性能会有更大提高，这样做更划算。

2.2.3 BIOS

对于基本的工作站，只需使用主要品牌的 BIOS (AWARD, AMIBIOS 和 Phoenix) 即可。AMI BIOS 存在一些问题，在使用有 PCI to PCI 桥接的 I/O 卡（如 Adaptec Quartet 4 端口的以太网卡）时会使情况复杂化。AWARD BIOS 给用户提供了比 AMIBIOS 或 Phoenix 更多的控制手段。在现代系统中，一个允许用户通过下载新版本升级 BIOS 的闪存 BIOS 芯片已经成为标准配置。

2.2.4 内存

对于典型的开发系统，64MB 内存是合理的选择。如果选择不运行 X Window 系统，那么在一台特殊用途的机器（比如用于调试设备驱动程序的“崩溃和烧毁”系统）上仅用 8MB 内存就可以工作。在 32MB 和 64MB 下编译内核所需的时间几乎一样，都少于一分半钟。在一个只有 8MB 内存的系统上，编译会需要更长的时间。

类似 Web 浏览器这样的多媒体应用软件在内存充足时会运行得更流畅，特别是在一边编译程序一边上网浏览的时候更是如此。所以如果只有 32MB 内存，则预期的性能会有所降低。类似地，如果要开发消耗大量内存的应用程序，可能会要求更多的内存。本书是在一台有 32MB 内存的机器上撰写的，而一位合作者的开发系统为了支持人工智能方面的工作拥有十倍于此的内存容量。

2.2.5 机箱和电源

选择机箱时应使之与主板的形状匹配，并能够提供足够的驱动器托盘和电源功率以满足需要。许多机箱生产厂商都重新改造了他们的 AT 机箱生产线以生产能够容纳 ATX 主板的机箱。如果你订购了一个 AT 机箱，收到的机箱可能已采用了新的 ATX 设计，在 I/O 挡片上预留了 AT 键盘和鼠标口。小型、中型或完全型塔式机箱是也是很好的选择。对于其他应用，可能需要服务器或机架式的设计。

注意： 不能在 ATX 机箱内使用 AT 电源，反之亦然，因为电源的接口不一样。

如果系统需要运行关键任务，需要注意某些电源在掉电后不能恢复正常。此时就需要使用小型冗余电源，其外形比普通的 ATX 或 PS/2 的电源稍大；一些面向高端系统的机箱，尤其是服务器机箱或机架，做了专门设计以容纳这种冗余电源或普通的 ATX 或 PS/2 电源。

2.3 用户交互硬件：视频、声音、键盘及鼠标

本节将要介绍的设备主要用于和用户交互。从制造商或其他来源可以获得支持显卡和显示器的大量信息。对于显示器来说，要取得 Linux 的支持所需的信息既使有也不多，而支持最新的显卡往往需要详细的编程信息。虽然有的显卡制造商已经开始提供其硬件产品的技术规范，甚至为其硬件编写 Linux 驱动程序，可大多数的显卡驱动程序仍然是由 Linux 社群本身提供的。所以，通常比较明智的做法是避免使用很新颖的卡，除非你能够事先确定有它的驱动程序。声卡和显卡一样，也需要相关的文档和具体编程支持，而音箱只需和声卡匹配就可以正常工作。

2.3.1 显卡

如果只运行文本模式的控制台，绝大多数 VGA 显卡都能工作的很好。但如果需要图形支持，就要选择被 XFree86、SVGAlib 或内核的普通视频帧缓冲功能支持的 VGA 显卡。

XFree86 是 X 窗口系统的一个免费的公开源代码实现，X 窗口系统是一个基于开放标准的窗口系统，所以运行在本机或网络上的图形应用程序都能访问显示器。对于开发工作站来说，XFree86 的支持通常是必要的，也肯定会带来方便。对于用于开发的工作站而言，必须而且只需支持 XFree86。在 <http://www.xfree86.org/> 上可以查阅有关信息，在 <http://www.suse.de/XSuSE> 上的 XFcom (XSUSE 的前身) 中可以查找到一些新视频设备的驱动程序。

SVGAlib 是一个在控制台以全屏方式运行图形程序的库，主要被某些游戏软件和图像浏览应用程序所使用，并且这类程序大部分都有 X 窗口系统或对应程序。遗憾的是，SVGAlib 应用程序需要超级用户的权限来访问视频硬件，所以通常在安装它们时要设置其 SUID 位，从而产生了潜在的安全问题。

OpenGL (及其前身 GL) 长期以来一直是 3D 图形事实上的标准，它提供了一套开放的 API，但直到最近才有了开放的、可自由获得的参考实现。OpenGL 的创造者 SGI (Silicon Graphics) 公司在 2000 年 2 月宣布它将在一种开放源代码许可证下发布 OpenGL。希望这一举措能让更多的显卡对 OpenGL 标准提供更好的硬件支持。在 SGI 的声明公布之前，需要 OpenGL 图形建模的 Linux 程序员只能使用 Mesa，Mesa 是 OpenGL 的一种开放源代码的实现，能够运行在 Linux 和其他许多平台下。对 3Dfx 的 Voodoo 卡的硬件加速的支持也已经可以获得了。在 <http://www.mesa.3d.org/> 上有关于 Mesa 的更多信息。Metrolink 公司得到 OpenGL 的许可并实现了一个商业产品，浏览 <http://www.metrolink.com/opengl/> 可以查阅有关信息。

内核本身就支持帧缓冲视频设备。帧缓冲视频设备创建的编程接口可以达到两个目的。首先，它让程序员编写的同一视频代码可以运行在多种处理器体系结构上，因为 API 是一

样的,而且是由内核驱动程序控制底层的硬件。帧缓冲代码也依赖于 VESA 视频标准——可以假定,如果制造商和程序员都遵循 VESA 标准,则任何兼容 VESA 的显卡都可以通过编程完成同样的功能。Framebuffer HOWTO 提供了有关帧缓冲视频的更多信息,它的地址是 <http://www.tahallah.clara.co.uk/programming/Framebuffer-HOWTO-1.1.html>。在 Intel 及其兼容平台上,vesalib 提供了对 VESA 2 兼容显卡的帧缓冲设备的支持。不幸的是,必须在启动时选择图形模式,如果想要改变这一模式,只能重启后再次选择。为什么呢? VESA 规范不允许在 CPU 处于保护模式时切换到视频模式,而 Linux 内核却是运行在保护模式下的。Linux 内核惟一没有运行在保护模式下的时刻是它刚启动的时候。

提示: 某些公司提供了用于 Linux 以及其他兼容 UNIX 操作系统的商用 X server。其中有 Metro Link (<http://www.metrolink.com/>), 它的产品叫 MetroX, 还有 XiGraphics (<http://www.xigraphics.com/>), 它的产品是 Accelerated-X。

AGP (Accelerated Graphics Port, 图形加速接口) 向处理器提供的访问视频存储器的接口要比 PCI 总线快四倍,而且提供的视频加速器能够更快速地访问保存在系统内存中的纹理映像。XFree86 支持绝大多数 AGP 图形卡。

提示: 要确定 X server 能支持的显卡,键入 server 的名称,比如 XF86-SVGA,后面跟着-showconfig。即: `$ XF86-SVGA -showconfig`

在 16 位色下支持 1280×1024 分辨率 (2.6MB) 需要至少 4MB 的显存,而支持分辨率 1600×1200 的 32 位色显示则需要 8MB 显存。某些 3D 游戏在有额外显存的情形下可以加快处理纹理映像或其他特性的速度。X server 本身使用少量内存维护字体缓存和扩展位图。

如果要配置一个大于实际物理尺寸的虚屏(这样,当把光标移动到实际屏幕的边缘时,物理的显示区域就在虚屏上滚动),则需要更多的显存。X server 还使用系统内存进行“后背存储”(backing store)。在用来做后背存储的内存中,X 重画了隐藏的部分窗口,在它们重新显示时起加速作用。如果在高分辨率或 16 位色以及 32 位色屏幕下工作,后背存储会对系统内存有额外的要求。

在 Linux 系统中安装支持高分辨率和高位色的显卡能带来很大好处。Linux 能同时轻松处理许多不同进程,因而用户希望拥有足够大的屏幕好同时观察多个窗口。一块支持 1280×1024 分辨率的显卡能够很好地满足这样的要求。好显卡的另一个优点是支持高位色。这不仅可以使较新的窗口管理器更平滑地工作,同时也对系统进行图形工作很有帮助。

当然,显示器必须能够支持显卡所支持的分辨率,否则就不能充分发挥显卡的功能。在购买一块显卡之前,要查看硬件兼容性列表确保 Linux 能够支持它。

2.3.2 显示器

几乎任何和显卡兼容的显示器只要知道三个关键参数:它的垂直和水平刷新率以及它的视频带宽,就能让它在 Linux 下正常工作。需要注意的一点是显示器并不是越大越好;关键是看在不牺牲质量的前提下的全屏像素数。例如,一台不贵的 17 英寸显示器在屏幕上每英寸可显示的像素数可以比一台昂贵的 20 英寸工作站显示器多。如果不能距离屏幕很近或者想坐得离显示器远些,就需要一台大尺寸的显示器,否则花比较少的钱买一台高质量

的小尺寸显示器，在较近的距离上可以获得相同或更高的显示质量。

如前节所述，显示器的选择和显卡有很大关系。最好首先对显示器的显示清晰度进行测试。点距是影响显示清晰度的一个重要因素。像素的点距越小越好。但通常点距越小价格越贵。

显示器的清晰度也取决于显卡。同样的显示器在不同的显卡下会有明显不同的显示效果。主要面向商业应用的显卡（比如 Matrox Millenium G200）一般都比面向游戏的显卡其图像的对比度更好，这是因为显卡往往针对 2D、3D 或对比度来优化，但很少三者都优化。

提示： 我建议显示器最好运行在 72Hz 以上的刷新率下。这样做能减少明显的屏幕闪烁，也就减轻了眼睛的疲劳。同时还要注意，有些显示器有意使图像在屏幕上下以极低的频率轻微抖动以保护屏幕。只要这种运动非常缓慢，眼睛就不会感到抖动（即使是头部的轻微运动速度也要比它快得多）。

Linux 下视频配置的灵活性让你在选择硬件时有了很大自由度。但要记住，你要花许多时间注视显示器，不要在这上面使用二流的产品。

2.3.3 声卡

Linux 支持大部分声卡，尤其是 SoundBlaster 兼容声卡（但并不直接支持所有宣称兼容的卡，其中的一部分需要软件模拟才能实现），老式的 ESS 声卡（688 或 1688），基于微软声音系统的声卡，以及 Crystal（Cirrus Logic）声卡。更为详细的信息，可以参考 Linux 硬件兼容性 HOWTO 文档，4 Front Technologies Web 站点（<http://www.4fronttech.com/>），或 Linux 内核资源（<http://metalab.unc.edu/Linux-source>）。4 Front Technologies 发售一种软件包，其中包含大量未与内核一同发布的声卡驱动程序。大多数新声卡似乎都是 PnP 设备。采用本章后面讨论的 ISAPnP 工具可以支持 PnP 卡。4Front 驱动程序对 PnP 声卡支持得非常好。

2.3.4 键盘及鼠标

现在仍不建议使用 USB 键盘和鼠标设备，详细的信息可以参考本章后面的“USB 与火线（IEEE1394）”节。除了不能支持键盘的某些额外特性，一般而言，普通的标准 AT 或 PS/2 风格接口的键盘口都能很好的工作。

键盘中内置的轨道球，光栅笔或轨道板一般都单独与串口或 PS/2 鼠标口相连，在软件支持的情形下，应当简单的把这些设备看作单独的鼠标。Linux 支持常用的 PS/2 和串口鼠标，其中包括使用 Microsoft 协议，Mouse System 协议或 Logitech 协议的鼠标。控制台的鼠标支持由 gpm 程序或 X Windows System 的 X server 提供。

许多其他的点设备，比如轨道球，光栅笔或轨道板在模拟普通鼠标并使用相同通信协议的情况下也能够被支持，但会丧失较新轨道板的某些特性，比如笔输入和较宽区域的特殊处理。许多 X 应用程序要求用户使用三键鼠标，因此 gpm 和 X server 提供了配置方法以用两键鼠标来模拟三键鼠标，此时，同时按下左右键就相当于按下了三键鼠标中的中键。

2.4 通信设备、端口及总线

本节叙述与通信通道有关的各种设备的信息。这些通道主要用于和其他计算机以及内部和外部的设备进行通信。

这里也会谈及连接处理器和扩展卡的高速总线。但是不会详细解释 ISA 总线或 PCI 总线——对于普通的 ISA 或 PCI 卡来说,只要存在支持它的驱动,它就能很好地工作。对 ISA 总线的即插即用设备和 PCMCIA 卡单独进行讨论,因为它们有特殊的问题。大多数 IDE 控制器都能工作,特别是那些连接到 IDE 硬盘的控制器更是如此;对于其他 IDE 设备,比如磁带机等,请参考本章中“存储设备”一节的内容。连接到并行口的设备,比如 Zip 驱动器和打印机都将分类进行讨论。

2.4.1 调制解调器

大多数调制解调器在 Linux 下都工作得很好。上述说法的一个著名例外是所谓的 WinModem。WinModem 类的调制解调器不是使用了专有的 Rockwell 协议接口 (Rockwell Protocol Interface, RPI),就是依靠软件部件来实现它们的功能。但是,应当注意昂贵的专业级(因特网服务提供商使用)产品和廉价的面向普通消费者产品之间是存在实质性差别的;几乎所有的调制解调器都能在高质量的电话线路上很好的工作,但对于质量低劣的线路,不同层次的产品其工作表现有明显的差别。这就是为什么对于同样的调制解调器,有人感到价廉物美,而有人却对其表现极为不满。在质量较差的通信连接中传输数据所需要的硬件结构要比通过高质量的连接传输的同样数据复杂得多,而传输所需要的时间也更多。

某些重要的开发者可能希望在办公室或家中用专线接入因特网。一些较为昂贵的调制解调器可以工作在租用线路的模式下。这种方式可以通过一条不受限的双线“干回路”(dry loop)与因特网建立一个 33.6Kbps 的专门(永久)连接。在不提供 ISDN 或 xDSL 的地区,这不啻为一种便利的解决方案。一条“干回路”是一条出租的电话线,但没有电压、没有振铃信号,也没有拨号音,它只是把两地永久地连接起来。有时它也被称为“小偷警报线”。在短距离的情况下,这种线路非常便宜。不幸的是,在电话公司业务办公室工作的一般人对此没有兴趣。支持“干回路”的调制解调器每个至少需要 200 美元。

由于只有带有数字电话接口的调制解调器才有可能用软件实现 56K 的应答方式,所以很难找到一对支持租用线路模式的 56K 调制解调器。即使 xDSL 一般用在调制解调器位于中心办公室一端的场合,在相对较短距离的“干回路”上仍然有可能运行 DSL。DSL 的一种变体 MVL 在长距离上工作得更好,并能在回路上提供 768Kbps 的速率。在某些地区,租 16 条线的费用大约为 \$13 000,但是在线数较少的情况下 xDSL 设备的价格却不够经济。如果能够在大量的线数上分摊资金投入,则 DSL 要比 ISDN 或 T1 线路经济得多。

如果想要支持拨入模式(应答模式)的 56K 连接,就需要提供数字线路接口的调制解调器。一般而言,ISP 使用昂贵的支持 T1 线路接口的调制解调器池,其中的每个调制解调器都支持数字接口,这种方式只有在需要支持的线数超过一定数量才比较经济。现在有一种调制解调器,既可以工作在普通模式,也可以作为 ISDN 的终端适配卡而工作在 56K 的应答模式上。

2.4.2 网络接口卡

许多人认为基于 Tulip 芯片的网卡是 Linux 系统上 PCI 以太网卡的最好选择。它价格低廉，传输速度快而且可靠，并有详细的文档。但最近新版的芯片（产于 1998 年年底~1999 年）存在一些轻微的兼容性问题。而更为安全的老的芯片已不再生产了，并且生产线也被卖给了竞争对手 Intel 公司，使得此类网卡极为短缺。这些问题似乎已经都解决了，但如果你正要购买老型号 Tulip 芯片的网卡应该注意一下。如需要在一台机器上安装多个以太网接口，可以使用 Adaptec Quartet 卡，它在一台机器上集成了 4 个以太网接口。

对于廉价的 10Mbps 的 ISA 网卡而言，廉价的 NE2000 克隆网卡通常工作得不错。这类网卡在传输数据时所占用的 CPU 时间要比其他设计复杂的卡略多一些，但是它能够在峰值下运行（但是不要指望仅通过一个诸如 FTP 的 TCP 连接来达到峰值，通常需要同时建立多个连接才能达到这一带宽）。

3Com 支持在 Linux 下使用他们的以太网卡，Crystal (Cirrus Logic) 也为他们的以太网控制器芯片提供 Linux 下的驱动程序。大多数的广域网卡制造商也将提供相关的 Linux 驱动程序。SDL, Emerging Technologies 以及 Sangoma 都提供了其产品的 Linux 驱动程序。

2.4.3 SCSI

Linux 支持绝大多数 SCSI 控制器，包括许多 RAID 控制器，主机适配器，几乎全部的 SCSI 硬盘以及大多数 SCSI 磁带驱动器和 SCSI 扫描仪，但不包括基于并口的主机适配器。Advansys 公司在 Linux 上支持该公司的 SCSI 适配器，他们提供的驱动程序随内核一同发行。随处可见的 Iomega Jaz Jet，其 PCI SCSI 控制器使用的就是 Advansys 的产品，而且的确物有所值。除非 SCSI 控制器及其驱动程序并且总线上的所有的慢速设备支持“disconnect reconnect”特性，否则不应把磁盘驱动器和一些诸如磁带驱动器或扫描仪等慢速设备连接到同一个 SCSI 总线上，不然的话，在磁带机倒带或扫描仪回车时，整个系统就有可能挂起 30 秒钟或更长时间。在 SCSI HOWTO 上有关于“disconnect reconnect”的详细信息。

警告： 要小心廉价的 SCSI 控制器，特别是那些不使用中断的 SCSI 控制器。典型情况下，这类 SCSI 随扫描仪一起捆绑销售，或者内置在某种声卡上。

2.4.4 USB 和火线 (IEEE1394)

对于 USB 以及火线的支持正处在开发之中。一种叫做 UUSB D 的软件包提供了对 USB 的支持。如果拥有一款被支持的 USB 控制器，还有可能使用 USB 鼠标，但在运行 X 之前需要下载并安装相应的代码。在撰写本书的时候，只有一定数量的 USB 键盘可以工作，但是数量还在增加。对 USB 的广泛支持被安排给了 2.4 版内核，在你阅读本书时它应该已经发布了。在一段时间里，除非要做修修补补的工作，否则还是保持使用老的接口（串行口、并行口和 PS/2）吧（当然，如果你打算在支持 USB 或火线上做些工作则例外）。

2.4.5 串行卡

Linux 支持主板上的以及额外的标准 PC 串口。但建议不要使用早期的不包含 16550A 或兼容的通用异步收发器 (Universal Asynchronous Receiver Transmitter, UART) 的串口，

因为它们既慢，输入缓冲又小。幸运的是，现在已经很难看到这类配件了。

Linux 也支持绝大多数智能多口串行卡，通常是直接由厂商提供相应的支持，其中包括 Cyclades, Equinox, Digi 和 GTEK 等公司。Equinox 提供了一种有趣的多重串口卡变种，这种卡使用一个外部底盘来支持 16 个 ISA 调制解调器（或对计算机而言与调制解调器类似的设备）。

大部分的多口预留串行卡能在 Linux 下工作，但除非在系统和 / 或端口负载很轻的前提下，否则不要在系统中安装太多的这类串行卡。Byterunner (<http://www.byterunner.com>) 支持其廉价的 2/4/8 口串行卡在 Linux 下的使用，与多数此类卡不同的是，Byterunner 的这些产品采用了高端配置，能够有选择的共享中断，并且支持常规的握手信号。

2.4.6 IRDA

Linux 最近开始支持红外数据关联（Infrared Data Association, IRDA）协议设备，因而可能还不很成熟。Linux 2.2 内核直接提供了对 IRDA 的支持，但使用时仍需要 irda utils 包。IRDA 项目的主页在 <http://www.cs.uit.no/Linux/irda/>。

2.4.7 PCMCIA 卡

Linux 对 PCMCIA 的支持已经有一段时间了，所以相当稳定。但对于特定的设备需要特定的驱动程序，如果你所使用的设备没有列在 Linux 发行版安装盘的 `/etc/pcmcia/config` 文件中，安装该设备可能会很困难。

2.4.8 ISA 即插即用设备

2.2.14 版内核对 ISA 即插即用（ISAPnP）的支持已经相当成熟了。但是传统上 Linux 上都是由工具程序 ISAPnP 提供对 PnP 的支持。和即插即用的名字相反，这些工具程序不能自动运行。但这个“缺点”的好处是减少了不可预见的、变化多端的运行结果，往往称其为“即插即祈祷”（Plug and Pray）更恰当。

使用 ISAPnP 时要先运行 `pnpdump` 创建一个配置文件的范例。这个文件包含了系统上每个 PnP 硬件可能的多种配置。然后手工编辑这个配置文件，选出一种特定的配置。要避免系统引导时需要使用 PnP 设备，比如磁盘控制器和网卡（对于那些从网络启动的系统而言）都不能是 PnP 设备。

2.5 存储设备

Linux 支持计算机消费市场上常见的许多存储设备。Linux 也支持企业级的设备，比如磁带库和大规模 RAID 阵列。Linux 支持的消费类设备包括差不多所有的硬盘驱动器和移动存储介质，比如 Zip、CD-ROM/DVD 和磁带驱动器。

2.5.1 硬盘

实际上 Linux 支持所有的 IDE、EIDE 以及 SCSI 硬盘驱动器。Linux 甚至还支持一些老式的 ST506 和 ESDI 硬盘控制器。Linux 也支持 PCMCIA 驱动器。Linux 支持的多种软硬件形式的 RAID (Redundant Array of Inexpensive Disks, 廉价磁盘冗余阵列) 配置, 它提供了快速、具有容错性的大容量存储硬盘。

2.5.2 可移动磁盘设备

最新版的 Linux 内核支持包括 Jaz、LS120、Zip 以及其他设备在内的移动存储介质。用这些设备作为启动设备会有些问题, 但它们都是优秀的长期存储介质。

2.5.3 CD-ROM/DVD

包括 IDE、SCSI 甚至许多老式专有接口的驱动器在内, 几乎所有的 CD-ROM 驱动器都能在 Linux 下很好地完成读取数据的功能。某些采用并行接口的光驱, 特别是 Micro Solutions 公司的 Backpack 驱动器, 也能工作。有些光驱不能用作音乐 CD 播放器, 因为在光驱的音频功能方面缺乏标准。只有很少的光驱能够读取“红皮书”(Red Book) 规范^①中说明的音频数据, 制订该规范是为了让计算机能够直接从音乐 CD 上读入数字音频数据, 再进一步用来复制、处理或传输。

Linux 支持许多类型的光盘组^②。eject 命令有个选项可以从光盘组中选出需要的光盘。cdrecord 命令用于向 CD-R 和 CD-RW 写入数据。在 <http://www.guug.de:8080/cgi-bin/winni/lsc.pl> 上的 UNIX CD-Writer 兼容性列表有大量关于兼容的光驱设备的信息。需要注意的是, 由于 CD-R 驱动器的限制, 写光盘的工作只能在系统负载非常轻, 或最好, 在专门的机器上进行。仅仅在数据流中的一个短暂中断也会破坏所有的数据。现在写 CD 的程序已经有了 GUI 前端, 其中有 BurnIT 和 xcdroast。

2.5.4 磁带备份设备

Linux 支持大量的磁带备份设备以及其他种类的移动介质。Linux 上的驱动程序能够支持 SCSI、ATAPI (IDE)、QIC、磁盘和某些并行接口。更好的解决方案是使用 SCSI DAT (Digital Audio Tape, 数字音频磁带), 尽管该设备的价格和一台廉价的 PC 相当, 因为 DAT 的解决方案能够保存大量的数据, 而且速度很快。

警告: 尽量不要在磁带设备上使用数据压缩; 因为读错误经常发生, 而一个错误就会使磁带上余下的整个部分变得不可读。

① 译者注: 音乐 CD、CD-ROM、CD-R、VCD、DVD 等方面的规范往往是由多家业内的公司联合制订的, 这些规范根据其封面的颜色而命名。

② 译者注: 这类光驱现在已经很少见了。1995-1996 年期间, 市场上比较常见的型号有 NEC4×4、Mitsumi4×4、Hitachi4×4 等。它们在外观上和普通光驱没有区别, 但可以连续放入四张光盘, 在操作系统中从功能上看等同于四个光驱。

2.6 外围设备

本节讨论的设备都是通常安装在系统机箱之外的可选外围设备。这些设备的驱动程序通常运行在用户空间而不是内核空间；也就是说，运行在内核之外，没有任何特殊的权限。

2.6.1 打印机

如果你拥有一台真正的 PostScript 打印机，可以跳过这一节的内容，因为 Linux 和 Linux 应用程序都直接支持 PostScript。另一方面，大多数人都没有 PostScript 打印机，如果你也是其中一位，那么可以用 Ghostscript 软件包 (<http://www.ghostscript.com/>) 支持你的 Linux 打印机。Linux 对于佳能 (Canon) 打印机的支持较差，因为历史上佳能公司不愿意提供技术文档。Linux 支持绝大多数惠普 (HP) 打印机 (以及模仿惠普的打印机)。Linux 对爱普生 (Epson) 打印机的支持也做得很好。要查看 Ghostscript 支持的打印机的完整列表，可以在命令行上键入：

```
$gs --help
```

gs 命令会列出长长的一份它所支持的打印机和输出设备的名单。

佳能 BJC-5000、BJC-7000 和惠普公司基于 PPA 的打印机都是打印机中的 winmodem，因为它们本机不安装字库而且没有硬件光栅——这类打印机要依赖计算机主机上的 CPU 来执行打印操作。但是，这对 Linux 系统不是一个问题，因为通常 Ghostscript 软件会被用作光栅和字库，而其他特性都用不到。

应该避免使用 HP720、HP820Cse 以及 HP1000 型号的打印机，因为它们没有 CPU，而且采用了一种非标准的有害的方式控制并行端口。Linux 可能支持某些利盟 (Lexmark) 喷墨打印机，但是剩下的许多同品牌喷墨打印机就只能用于 Windows 了。但是，利盟的 Optra R+ 激光打印机带有以太网接口，在 Linux 下工作得很好。它的固件 (软件固化嵌入到硬件中) 支持 LPD 协议，所以可以把它简单地设置为一台远端 LPD 设备。

Ghostscript 能在任何有足够资源产生光栅数据的操作系统上运行，它可以通过一个驱动程序 (或 PBM 转译器) 来支持几乎所有类型的计算机系统，包括 UNIX 兼容系统，MacOS，OS/2，Windows 3.1，Windows 95，Windows 98 和 Windows NT 等。此外，Ghostscript 能取代这些操作系统的本机光栅数据生成器，或者与之共存，或者与几乎所有这些操作系统的打印子系统集成。

提示： 在 Linux 系统下与打印有关的详细信息可以参见 Linux 的 Printing HOWTO。

2.6.2 扫描仪

虽然 SANE (Scanner Access Now Easy) 软件支持 20 余家厂商的近 100 种不同型号的扫描仪，但是 Linux 对扫描仪的支持仍显不足。总体来说，扫描仪制造商应当为缺乏 Linux 支持的状态负责：他们既不提供自己产品的驱动程序，也不公开编写扫描仪驱动程序所需的技术资料。

2.6.3 数字相机

Linux 上有许多支持手持式数字相机的程序。支持用闪存或磁盘来存储标准 JPEG 图像的相机也应该支持用这些介质来传输图像数据。一种称为 gPhoto 的新的应用程序能够支持大约十种不同品牌的数字相机。SANE 库也支持一些数字相机。

另外，因特网上有 Frame Grabbers、TV tuners 以及流行的 Quickcam 相机的软件驱动程序。参考 HOWTO 的硬件兼容性列表的相关部分可以找到指向这些资源的链接。

2.6.4 家居自动控制设备

如果你是一个真正的发烧友，那么你会喜欢上下面这个小玩意，因为用它能控制现实世界，而且还是工作在 Linux 下的。有两个程序可以控制 X10 CM11A（通常作为 CK11A 套件的一部分出售）计算机接口模块。X10 系统能够通过家庭或办公室的电力线路传送控制电源的承载电流信号，来远程控制家电的打开与关闭。X10 承载电流协议受专利保护，但同时有详细的文档；因特网上能找到 X10 系统与计算机接口的相关文档。

2.7 完备型系统

为数不少的计算机厂商专门提供预装 Linux 的系统。VA Linux 系统公司和 Linux Hardware Solutions 公司是其中的佼佼者。在 <http://www.linux.org/vendors/systems.html> 能找到销售整套 Linux 系统的厂商的完整列表。

Rebel.com (<http://www.rebel.com>) 收购了 Corel 公司生产 Netwinder 系列产品的部门，现在他们销售基于 Linux 的桌面和服务器的全线产品，这些系统都采用了 StrongARM CPU。它们都是面向瘦客户机和 Web 服务器市场的系统，价格相当便宜，但它们形成了理想的低端开发系统。Cobalt Networks 提供的 Qube 是一种基于 Linux 的紧凑型服务器，它使用 MIPS 处理器。SGI 也在其某些型号的工作站上预装 Linux。

包括戴尔、IBM、惠普以及康柏在内的几大 PC 生产商现在或准备在自己某些型号的工作站和服务器的上预装 Linux。不难想像，以后几个月内会有更多的厂商加入这个阵营。

2.8 便携系统

支持便携系统需要保持机警的头脑，因为这类系统开发周期短，经常使用很新的半导体部件，而且制造商也很少会提供技术资料。尽管如此，因特网还是存在大量信息涉及在数百种便携机上如何使用 Linux 的内容。Linux 便携机主页 <http://www.cs.utexas.edu/users/kharker/linux-laptop/> 上有关于如何在便携机上使用 Linux 的最全面的信息资源。这个站点包含了大量文章，并给出了数百种已知能够运行 Linux 的便携机型号。Linux 也支持掌上电脑，比如菲利普 Nino、苹果 Newton 和 3Com 的 PalmPilot 等。

注意： Linux 支持多种电子记事本。3Com 的 PalmPilot 最流行，Linux 对它的支持也最好。

Linux 支持自动电源管理 (Automatic Power Management, APM)，但比较简陋。目前已知的问题有挂起/恢复特性不能正常工作，在 X Window 系统下正确恢复图形模式困难。需要专门保留一个 DOS 分区以保证硬盘挂起功能正常工作。有的便携机不允许磁盘和 CD-ROM 同时工作，即使绝大多数新型号的便携机都支持从 CD-ROM 启动，这还是有可能会给安装带来一定的困难。

2.9 开发工具软件

迄今为止的讨论都集中在如何为一个开发系统选择硬件的话题上。本节考虑 Linux 主机软件方面的话题。大多数 Linux 发布版本都提供了一个安装选项能够装好一个完整的开发环境。这个环境包括编程库、头文件、汇编器、分析器、编译器、链接器、调试器、文本编辑器和编程工具。总体而言，如果在安装 Linux 系统时选择了 Linux 厂商提供的选项，那么可以建起一个实用的开发环境。如果没有特意安装开发工具，本节会全面介绍需要安装的软件。

2.9.1 关键库和头文件

什么构成了“关键库和头文件”？许多对编程的支持，尤其是系统调用，都来自于内核的源代码文件，所以应该安装好内核的源代码文件和头文件。在大多数 Linux 系统上，这些文件都位于 `/usr/src/linux` 目录下。另外还要有 C 的开发库，`glibc`。在采用 RPM 包管理系统的 Linux 系统上，C 开发包典型的名字是 `glibc-devel-something.rpm`。

其他要安装的库和头文件取决于开发工作的内容。比如，如果希望进行大量图形程序方面的开发工作，那么要安装 X 开发库，还有目前使用的桌面环境 GNOME 或 KDE 的库（和头文件）。如果所用的 Linux 版本不带 GNOME 的库文件，可以从 <http://www.gnome.org/> 处下载。类似地，KDE 的开发包可以从 <http://www.kde.org/> 处取得。

2.9.2 调试器

除非你能写出没有错误的代码，否则就需要一个源代码调试器。这个调试器一般是 GNU Debugger，即 `gdb`。另一个，也是和 KDE 最兼容的调试工具，`kdbg` 是 `gdb` 在 KDE 上的图形前端。`ddd`（代表 Display Data Debugger）和 `xgdb` 都是 `gdb` 普通的 X 前端。

2.9.3 编程工具

大量的程序都可以归为这一类工具。比如，`indent` 能够按照许多预先定义好的或者自定义的标准调整源代码以及代码缩进的格式以达到所需的风格。`tags` 程序（它在 `emacs` 上的对应程序，`etags`）生成的文件能够增强编辑器浏览和分析源代码的能力。所有这些程序都可以从 GNU 的 Web 站点 <http://www.gnu.org/> 下载。如果你希望在编译源代码之前做个快速语法检查，或者如果你希望进行更完整的代码分析，`lclint` 程序是完成此项任务最合适的

工具，它可以从 <http://lclint.cs.virginia.edu/> 取得。它能检查代码，找出大量问题，包括未声明的变量和函数、可能的内存破坏（memory corruption），当然它能检查的问题不仅仅就这几种。另一类工具检查潜在的内存分配问题。网上这类工具中最好的资源是“动态存储分配和内存管理信息库”（Dynamic Store Allocation and Memory Management Information Repository），它的 Web 地址为 <http://www.cs.colorado.edu/~zorn/DSA.html/>。

2.9.4 文本编辑器

vi 和 emacs 可能是 Linux（和 UNIX）上最知名和最受程序员喜爱的两种编辑器了。通常用到的 vi 实际上是 vim（代表 VI iMproved）。它支持语法加亮功能，能够跟踪来自编译器和调试器的调试和出错信息，支持 tag 文件，带有大量工具宏，还具有许多其他在编写代码时起辅助作用的功能。emacs，用一句话来形容，是一种伟大的操作系统，但 Linux 系统与之相比拥有更多的程序。严肃地说，即便 emacs 难以学会，但它可能包含了对程序员最丰富的支持功能。它不但具有 vi 全部的特性，而且还有增加：它既具有模仿几种调试器的模式，集成了对 RCS 和 CVS（源代码控制系统）的支持，又能够自动施加多种源代码格式，另外，通过其自身版本的 LISP 编程语言 ELISP 可以对它进行极为灵活多样的配置。

但是这些还不是你仅有的选择。Kdevelop 以其在编写、编译、调试和浏览源代码方面的功能，成为能够和微软的 Visual Studio 相媲美的集成开发环境（Integrated Development Environment, IDE）。另一种流行的 IDE Nedit 不但高度可配置，而且学会使用它极为简单。Metrowerks 公司推出的极为流行的 Code Warrior 也有在 Linux 平台上的版本（如果你不介意使用或购买商业软件的话）。

还有大量其他编辑器，比如 joe、jed、pico 和 uemacs，虽然没有上面介绍的几种流行，但也有它们各自的追随者。

2.10 小 结

本章详细论述了在建立一个软件开发系统时会遇到的问题。它讨论了主要的硬件种类，注明了 Linux 支持的计算机部件，并且提供了一些选择硬件时应牢记的原则。从这些讨论中能够总结出一个结论，在 Linux 像 Windows 一样得到硬件厂商良好支持之前，应该保留通过查看因特网上丰富的信息资源来确定硬件是否被支持的好习惯。

另外，本章还总体介绍了一个 Linux 开发系统的软件构成。当大多数 Linux 发布版本都有安装选项能够装好一个完善的软件开发环境的时候，读者应该对组成一个开发系统的关键部件，比如编辑器、编译器和工具程序等有所了解。

第3章 使用 GNU CC

GNU CC(常常称为 GCC)是 GNU 项目的编译器套件。它能够编译 C、C++和 Objective C 语言编写的程序。GCC 能够支持多种不同的 C 语言变体,比如 ANSI C 和传统(Kernighan 和 Ritchie, K&R) C。此外, GCC 在 g77 的帮助下也能够编译 Fortran 程序,而用于支持 Pascal, Modula 3, Ada 9X 以及其他语言的编译器前端也在开发中。由于 GCC 是 Linux 开发的基础,本章将对其在一定程度上进行深入介绍。本章乃至本书的例子,除非特别注明,采用的 GCC 版本都是 2.91.66。

注意: 如果你涉猎于 Linux 开发社群的时间足够长,那么一定听说过或读到过有关另一种编译器 egcs 的消息, egcs 代表 Experimental (Enhanced) GNU Compiler Suite。开发 egcs 的目的是为了让编译器比 GCC 更高效,它对编译器的开发工作也比 GCC 更活跃。它基于 GCC 代码并紧紧跟踪 GCC 的版本。长话短说, 1999 年 4 月维护 GCC 的自由软件基金会(Free Software Foundation, FSF)指派 egcs 的管理委员会作为 GCC 的正式维护者。与此同时, GCC 改名,从原来代表 GNU C Compiler 变为代表 GNU Compiler Collection。另外, egcs 和 GCC 的基础代码进行合并,结束了 GCC 基础代码中的一个长分支,并且修正了许多错误,加入了许多增强特性。所以说, egcs 和 GCC 从各方面看都是同一程序。

3.1 GNU CC 特性

使用 GCC, 程序员能够对编译过程有更多的控制。编译过程分为 4 个阶段:

- 预处理
- 适当编译
- 汇编
- 链接

而程序员可以在编译的任何阶段结束后停止整个编译过程以检查或使用编译器在该阶段的输出信息。你可以控制嵌入在生成的二进制执行文件中调试代码的数量和类型;同时,和其他编译器一样, GCC 也能优化执行代码。GCC 能够在生成调试信息的同时对代码进行优化,但笔者强烈反对使用这一特性,因为很难对经过优化的代码进行调试:由于静态变量可能被取消,循环也有可能被展开,所以优化后的代码与源代码已经不是行行对应了。

GCC 有 30 多个警告和 3 个一般警告级。同时, GCC 是一个交叉平台编译器,所以能够在当前 CPU 平台上为不同体系结构的硬件系统开发软件。最后, GCC 对 C 和 C++进行了大量扩展,这些扩展中的大部分能够提高程序的执行效率,或有助于编译器进行代码优

化，或使得编程变得更加容易，然而这是以降低可移植性为代价的。在本章中将会提及在内核头文件中出现的那些最常用的扩展，但笔者建议在编程中避免使用这些扩展特性。

3.2 教学示例

在开始深入了解 GCC 之前，下面这个小例子能够帮助读者立即开始高效地使用 GCC。为了达到这个教学目的，我们将使用程序清单 3.1 中给出的示例程序。

程序清单 3.1 示范 GCC 用法的示例程序

```
/*
 * hello.c - Canonical "Hello, world!" program
 */
#include <stdio.h>

int main(void)
{
    printf("Hello, Linux programming world!\n");
    return 0;
}
```

在命令行上键入以下命令编译和运行这段程序：

```
$ gcc hello.c -o hello
$ ./hello
Hello, Linux programming world!
```

第一行命令告诉 GCC 对源程序 `hello.c` 进行编译和链接，并使用 `-o` 参数指定创建名为 `hello` 的可执行程序。第二行命令执行 `hello` 这个程序，第三行是程序的执行结果。

整个过程看上去仿佛是一气呵成的，但是在这段时间里还发生了许多读者并没有看到的事情。GCC 首先运行预处理程序 `cpp` 来展开 `hello.c` 中的宏并在其中插入 `#include` 文件所包含的内容；然后把预处理后的源代码编译成为目标代码；最后，链接程序 `ld` 创建一个名为 `hello` 的二进制文件。图 3.1 图示了编译的全过程。

在编译过程中，可以通过手工操作重新创建这些步骤，以逐步执行编译过程。第一步是运行预处理器。使用 `-E` 选项告诉 GCC 在预处理后停止编译过程：

```
$ gcc -E hello.c -o hello.cpp
```

此时查看 `hello.cpp` 会发现 `stdio.h` 的内容确实都插到文件里去了，而其他应当被预处理的标记也做了类似处理。图 3.2 显示了 `hello.cpp` 文件从 894 行开始的部分内容。

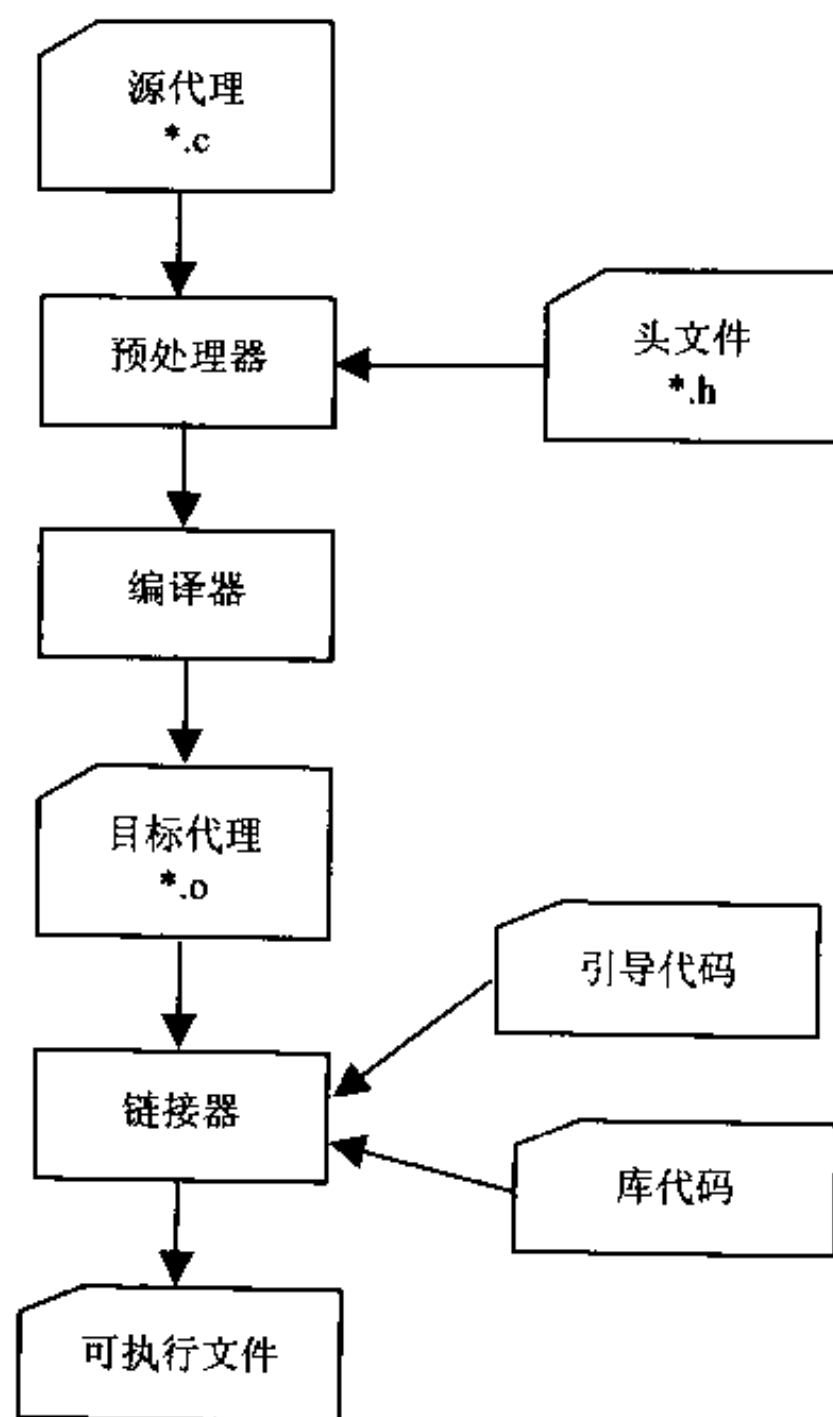


图 3.1 编译过程

```
# 29 "/usr/include/bits/types.h" 2 3
typedef unsigned char __u_char;
typedef unsigned short __u_short;
typedef unsigned int __u_int;
typedef unsigned long __u_long;

__extension__ typedef unsigned long long int __u_quad_t;
__extension__ typedef long long int __quad_t;
# 48 "/usr/include/bits/types.h" 3
typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
typedef unsigned int __uint32_t;

__extension__ typedef signed long long int __int64_t;
__extension__ typedef unsigned long long int __uint64_t;
```

图 3.2 预处理后的 hello.c 文件

注意：这段文本确切的位置可能随系统的不同而稍有变化。

下一步是将 hello.cpp 编译为目标代码。可使用 GCC 的 -c 选项来完成：

```
$ gcc -x cpp-output -c hello.cpp -o hello.o
```

本例中不必指定输出文件名，因为编译器通过将源文件名的.c 后缀改为.o 创建了目标文件名。-x 选项告诉 GCC 从指定的步骤开始编译，在本例中也就是编译器处理后的源代码（cpp-output）。

GCC 是怎样知道如何处理某种特殊类型的文件呢？它是依靠文件的扩展名来决定如何正确处理该文件的。表 3.1 列举了最常用的文件扩展名及其含义。

表 3.1 GCC 对文件扩展名的解释

扩展名	类型
.c	C 语言源代码
.C,.cc	C++语言源代码
.i	预处理后的 C 源代码
.ii	预处理后的 C++源代码
.S,.s	汇编语言源代码
.o	编译后的目标代码
.a,.so	编译后的库代码

最后，链接目标文件，生成二进制代码：

```
$ gcc hello.o -o hello
```

幸运的是，我们可以只用前面用过的 `gcc hello.c -o hello` 这一条缩略语法来执行所有这些步骤，这里分步执行只是说明在必要的时候可以在编译的任何阶段停止或开始编译过程；比如，在编译库程序时就需要分步执行编译程序，在这种情况下，只须生成目标文件，所以最后的链接是不必要的。当程序的包含文件（即 `#include` 文件）互相冲突或包含文件与程序冲突时，可能也需要分步编译以确定是哪个文件引起了冲突。

大多数 C 程序是由多个源代码文件组成的，所以必须将源代码文件编译成目标代码后才能进行链接。使用 GCC 很容易做到这一点。举例来说，假定 `hello.c` 使用了来自 `helper.c` 代码（参见程序清单 3.2 和 3.3）。程序清单 3.4 显示了修改后的 `hello` 程序的源代码，修改后的 `hello.c` 改名为 `howdy.c`。

程序清单 3.2 `howdy.c` 用到的 `helper` 的代码

```
/*
 * helper.c - Helper code for howdy.c
 */
#include <stdio.h>

void msg (void)
{
    printf("This message sent from Jupiter.\n");
}
```

程序清单 3.3 helper.c 的头文件

```

/*
 * helper.h - Header for helper.c
 */
void msg(void)

```

程序清单 3.4 修改后的 hello 程序

```

/*
 * howdy.c - Modified "Hello,World!" program
 */
#include <stdio.h>
#include "helper.h"

int main(void)
{
    printf("Hello,Linux programming world!\n");
    msg();
    return 0;
}

```

要正确编译 howdy.c, 可以使用下面的命令行:

```
$ gcc howdy.c helper.c -o howdy
```

同前面介绍的一样, GCC 也分别执行了预处理-编译-链接三个步骤。这一次, 在链接生成二进制程序 howdy 之前, GCC 首先生成了每个源代码文件的目标文件。键入类似这样的长命令行会比较烦人。在第 4 章中将介绍怎样解决这个问题。下一节将介绍 GCC 的命令行选项。

3.3 常用命令行选项

GCC 可以接受的命令行选项长达数页, 所以在表 3.2 中只列出了最常用的部分。

表 3.2 GCC 命令行选项

选项	说明
-o FILE	指定输出文件名, 在编译为目标代码时, 这一选项不是必须的。如果 FILE 没有指定, 默认文件名是 a.out
-c	只编译不链接
-DFOO=BAR	在命令行定义预处理宏 FOO, 其值为 BAR
-IDIRNAME	将 DIRNAME 加入到包含文件的搜索目录列表中
-LDIRNAME	将 DIRNAME 加入到库文件的搜索目录列表中
-static	链接静态库, 即执行静态链接默认情况下 GCC 只链接共享库

(续表)

选项	说明
-lFOO	链接名为 libFOO 的函数库
-g	在可执行程序中包含标准调试信息
-ggdb	在可执行程序中包含只有 GNU debugger (gdb) 才能识别的大量调试信息
-O	优化编译过的代码
-ON	指定代码优化的级别为 N, $0 \leq N \leq 3$, 如果未指定 N, 则默认级别为 1
-ansi	支持 ANSI/ISO C 的标准语法, 取消 GNU 的语法扩展中与该标准有冲突部分(但这一选项并不能保证生成 ANSI 兼容的代码)
-pedantic	允许发出 ANSI/ISO C 标准所列出的所有警告
-pedantic-errors	允许发出 ANSI/ISO C 标准所列出的所有错误
-traditional	支持 Kernighan & Ritchie C 语法 (如用旧式语法定义函数)。如果读者不知道这个选项的含义, 也没有关系
-w	关闭所有警告, 建议不要使用此项
-Wall	允许发出 GCC 能提供的所有有用的警告。也可以用 -W{warning} 来标记指定的警告
-Werror	把所有警告转换为错误, 以在警告发生时中止编译过程
-MM	输出一个 make 兼容的相关列表
-v	显示在编译过程的每一步中用到的命令

前面我们已经用过了 -c 选项, 知道了它的工作方式, 这里有必要讨论一下 -o 选项的用法。-o FILE 告诉 GCC 把输出定向到 FILE 文件中而不论实际上有没有生成输出数据。如果不指定 -o 选项, 对于名为 FILE.SUFFIX 的输入文件, 其生成的可执行程序名是 a.out, 目标代码文件是 FILE.o, 汇编代码在 FILE.s, 而预处理输出则定向到标准输出。

3.3.1 函数库和包含文件

正如你在表 3.2 中所看到的那样, -I{DIRNAME} 选项可以向 GCC 搜索包含(include)文件的路径中增加新目录。例如, 如果已经在 /home/fred/include 下保存了自定义的头文件, 那么为了让 GCC 能够找到它们, 可按下面的例子使用 -I 选项:

```
$ gcc myapp.c -I /home/fred/include -o myapp
```

-L 选项对库文件起的作用和 -I 选项对头文件起的作用类似。如果使用了不在标准位置的库文件, -L{DIRNAME} 选项告诉 GCC 把 DIRNAME 添加到库文件搜索路径里, 要保证 DIRNAME 比标准位置先被搜索。

假设读者需要测试一个新的编程库 libnew.so, 当前它保存在 /home/fred/lib 下 (.so 是共享库文件的标准扩展名, 这在第 10 章中有详细的说明)。为了链接库文件, GCC 命令行应与下面类似:

```
$ gcc myapp.c -L/home/fred/lib -lnew -o myapp
```

-L/home/fred/lib 结构让 GCC 先在 /home/fred/lib 下查找库文件, 然后再到默认的库文件搜索路径下进行查找。-l 选项使得链接程序使用指定的函数库中的目标代码, 也就是本例

中的 `libnew.so`。把函数库命名为 `lib{名字}` 是 UNIX 的约定，与许多其他编译器一样，GCC 也遵循此约定。如果忘了使用 `-l` 选项，则与库的链接将失败，并且 GCC 产生错误，说明程序中引用了未定义的函数名。

自然，你可以一起使用所有这些选项——实际上，除了最小的程序之外，对于大多数程序来说，这种做法相当常见(而且通常也是必要的)。也就是说，命令行：

```
$ gcc myapp.c -L/home/fred/lib -I/home/fred/include -lnew
-o myapp
```

告诉 GCC 链接 `libnew.so`，在 `/home/fred/lib` 中查找 `libnew.so`，以及在 `/home/fred/include` 中查找任何非标准的头文件。

默认情况下，GCC 使用共享库进行链接，所以在需要链接静态库时必须使用 `-static` 选项来保证只使用静态库。下面的例子生成了链接了静态库 `ncurses` 的可执行文件（第 23 章和第 24 章将讨论使用 `ncurses` 来进行用户界面编程）：

```
$ gcc cursesapp.c -lncurses -static -o cursesapp
```

在链接静态库时，可执行程序的大小比链接共享库要大很多。那为什么要使用静态库？通常的原因是保证程序在任何情况下都能运行——使用共享库时，程序所使用的代码是在运行时动态链接，而不是在编译时静态链接，因此，如果所需要的共享库没有为用户系统中安装，运行就会失败。

Netscape 浏览器就是一个很好的例子。Netscape 需要使用 Motif（一个昂贵的 X 编程工具包）。但是多数的 Linux 用户不能承担在系统中安装 Motif 的费用，所以 Netscape 实际上在用户的系统中安装了两个版本的浏览器，一个链接共享库（`netscape-dynMotif`），另一个链接静态库（`netscape-statMotif`）。而“可执行”`netscape` 实际上是一个 shell 脚本，检查系统中是否安装了 Motif 共享库以决定需要启动哪一个二进制程序。

3.3.2 警告和出错消息选项

GCC 含有完整的出错检查，警告生成功能及其命令行选项，包括 `-ansi`，`-pedantic`，`-pedantic-errors` 和 `-Wall`。`-pedantic` 选项允许 GCC 发出遵循严格的 ANSI/ISO 标准 C 语法的所有警告，所有使用被禁止的扩展语法（如 GCC 对 C 语法的扩展）的程序都将被拒绝。`-pedantic-errors` 有相似的作用，区别之处在于此时产生的是错误而不是警告且停止编译。`-ansi` 取消 GNU 扩展中所有与标准语法冲突的部分。但是，并不能保证在所有这些选项都被设置的情况下编译成功的程序就百分之百的遵循 ANSI/ISO 标准。

考虑程序清单 3.5 所列的程序，该程序写得很差，在声明时 `main` 函数的返回值是 `void`，而实际上返回 `int` 值，并且在其中使用了 GNU 语法扩展，即用 `long long` 来声明 64 位整数。而且在 `main` 函数终止前没有调用 `return` 语句。

程序清单 3.5 不符合 ANSI/ISO 的源代码

```
/*
 * pedant.c - use -ansi, -pedantic or -pedantic-errors
 */
#include <stdio.h>
```



```
void main(void)
{
    long long int i = 01;
    printf("This is a non-conforming C program\n");
}
```

使用 `gcc pedant.c -o pedant` 这一命令时，编译器会警告 `main` 函数的返回类型无效：

```
$ gcc pedant.c -o pedant
pedant.c: In function 'main':
pedant.c:7: warning: return type of 'main' is not 'int'
```

现在给 GCC 加上 `-ansi` 选项：

```
$ gcc -ansi pedant.c -o pedant
$ gcc pedant.c -o pedant
pedant.c: In function 'main':
pedant.c:7: warning: return type of 'main' is not 'int'
```

GCC 再次给出了同样的警告信息，并忽略了无效的数据类型。在这里需要知道，`-ansi` 只是强制 GCC 生成标准语法所要求的诊断信息，但并不保证没有警告的程序就是遵循 ANSI C 标准的。GCC 就没有发现本例中对 `main` 函数故意的错误声明和错误的数据类型。

现在使用 `-pedantic` 选项：

```
$ gcc -pedantic pedant.c -o pedant
pedant.c: In function 'main':
pedant.c:8: warning: ANSI C does not support 'long long'
pedant.c:7: warning: return type of 'main' is not 'int'
```

虽然发出了警告，代码还是编译通过了。但是这次编译器至少注意到了无效的数据类型。可是使用 `-pedantic-errors` 选项后代码就不能编译通过了。GCC 在发出错误信息后停止编译：

```
$ gcc -pedantic-errors pedant.c -o pedant
pedant.c: In function 'main'
pedant.c:8: ANSI C does not support 'long long'
pedant.c:7: warning: return type of 'main' is not 'int'
$ls
hello.c helper.c helper.h howdy.c pedant.c
```

再强调一遍，`-ansi`、`-pedantic` 以及 `-pedantic-errors` 编译选项并不能保证被编译的程序的 ANSI/ISO 兼容性，它们仅被用来帮助程序员离这个目标更近。GCC 的 `info` 文件中关于 `-pedantic` 的说明很有启发意义：

“设立这个选项并不希望有太大用途；它的存在只是为了让那些老是说 GNU CC 不能完全支持 ANSI 标准的吹毛求疵者（pedants）满意。有些用户试图用 ‘`-pedantic`’ 来检查程序是否严格遵循 ANSI C 标准。他们很快就会发现这样做并不能达到目的：这样做能发现一些非 ANSI 标准的情况，但并不能找出全部非 ANSI 的用法——而只是 ANSI C 要求进行编译器诊断的那些情况才能被发现。”

除了-ansi、-pedantic 和-pedantic-errors 编译器选项之外，GCC 还有许多其他的编译选项能够产生有用的警告信息。它们中间最有用的就数-Wall 选项了，它能让 GCC 发出多种警告信息，警告信息虽然不是错误信息，但是有潜在的危險，或者可能就是错误。下面的例子给出了在 pedant.c 上使用-Wall 时出现的结果：

```
$ gcc -Wall pedant.c -o pedant
pedant.c:7: warning: return type of 'main' is not 'int'
pedant.c: In function 'main':
pedant.c:8: warning: unused variable 'i'
```

注意，这次 GCC 找出了从未使用过的变量 i。显然，这不是个错误，但却反映出程序写得很差。

-Wall 系列选项的另一个极端是-w 选项，它能关闭所有的警告信息。-W{warning}选项的作用是打开 warning 所指出的用户感兴趣的特殊警告信息，比如隐式函数声明（-Wimplicit-function-declaration）或者隐式声明返回类型的函数（-Wreturn-type）。前者的作用在于它提示出定义了一个函数却没有事先声明或者忘记了包含适当的头文件。后者的作用在于指出声明的函数可能没有指定它的返回类型，此时默认的返回类型为 int。表 3.3 列举了 GCC 提供的的一些警告信息，它们对抓出常见的编程错误很有帮助。

提示： 如果只想检查语法而实际上不做任何编译工作，那么可以调用带有 -fsyntax-only 参数的 GCC。

表 3.3 GCC 警告选项

选项	说明
-Wcomment	如果出现了注释嵌套(在第一个/*之后又出现了第二个/*)则发出警告
-Wformat	如果传递给 printf 及其相关函数的参数和对应的格式字符串中指定的类型不匹配则发出警告
-Wmain	如果 main 的返回类型不是 int 或者调用 main 时使用的参数数目不正确则发出警告
-Wparentheses	如果在出现了赋值(例如，(n=10)) 的地方使用了括号，而那里根据前后关系推断应该还是比较(例如，(n==100))而非赋值的时候，或者如果括号能够解决运算优先级问题的时候，发出警告
-Wswitch	如果 switch 语句少了它的一个或多个枚举值的 case 分支(只有索引值是 enum 类型时才适用)则发出警告
-Wunused	如果声明的变量没有使用，或者函数声明为 static 类型但却从未使用，则发出警告
-Wuninitialized	如果使用的自动变量没有初始化则发出警告
-Wundef	如果在 #if 宏中使用了未定义的变量作判断则发出警告
-Winline	如果函数不能被内联则发出警告
-Wmissing-declarations	如果定义了全局函数但却没有在任何头文件中声明它，则发出警告
-Wlong-long	如果使用了 long long 类型则发出警告
-Werror	将所有的警告转变为错误

正如你所看到的那样，GCC 能够抓出许多常见而烦人的编程失误。程序清单 3.6 列举了一些典型的代码错误；在代码后面举出的 GCC 命令行说明了能够起作用的-W{warning} 选项。

程序清单 3.6 常见的编程错误

```
/*
 * blunder.c - Mistakes caught by -W{warning}
 */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int i, j;
    printf("%c\n", "not a character"); /* -Wformat */
    if (i = 10)                        /* -Wparentheses */
        printf("oops\n");
    if(j != 10)                        /* -Wuninitialized */
        printf("another oops\n");
    /* /* */                          /* -Wcomment */
    no_decl();                         /* -Wmissing-declaration
*/
    return(EXIT_SUCCESS);
}

void no_decl(void)
{
    printf("no_decl\n");
}
```

注释中给出了 GCC 会发出的警告。首先尝试只用简单的命令行，不带任何警告选项来编译这个程序。结果如下：

```
$ gcc blunder.c -o blunder
blunder.c:27: warning: type mismatch with previous implicit
declaration
blunder.c:21: warning: previous implicit declaration of 'no_decl'
blunder.c:27: warning: 'no_decl' was previously implicitly declared
to return 'int'
```

正如你所看到的那样，在默认的出错检查模式下，GCC 只发出了和隐式声明 no_decl 函数有关的警告。它忽略了其他潜在的错误，这些错误包括：

- 传递给 printf 的参数类型（一个字符串）和格式化字符串定义的类型（一个字符）不匹配。这会产生一个-Wformat 警告。
- 变量 i 和 j 使用前都没有初始化。它们中的任何一个都会产生-Wuninitialized 警告。

- 在根据前后关系推断应该对变量 `i` 的值做比较的地方，却对变量 `i` 进行赋值。这应该会产生一个 `-Wparentheses` 警告。
- 在注释嵌套开始的地方应该会产生一个 `-Wcomment` 警告。

首先，看看 GCC 是否能够抓出 `printf` 语句中的类型不匹配：

```
$ gcc -Wformat blunder.c -o blunder
blunder.c: In function 'main':
blunder.c:11: warning: int format, pointer arg (arg 2)
blunder.c: At top level:
blunder.c:27: warning: type mismatch with previous implicit
declaration
blunder.c:21: warning: previous implicit declaration of 'no_decl'
blunder.c:27: warning: 'no_decl' was previously implicitly declared
to return 'int'
```

你可以看到，诊断输出的前 3 行表明 GCC 抓出了 `printf` 调用中类型不匹配的问题。接下来，测试 `-Wparentheses` 和 `-Wcomment` 选项：

```
$ gcc -Wparentheses -Wcomment blunder.c -o blunder
blunder.c:19: warning: '/*' within comment
blunder.c: In function 'main':
blunder.c:13: warning: suggest parentheses around assignment used
as truth value
blunder.c: At top level:
blunder.c:27: warning: type mismatch with previous implicit
declaration
blunder.c:21: warning: previous implicit declaration of 'no_decl'
blunder.c:27: warning: 'no_decl' was previously implicitly declared
to return 'int'
```

和预期的结果一样，GCC 发出了在代码第 19 行有明显的注释嵌套的警告和代码第 13 行有可能错误地赋值的警告。

最后，测试 `-Wuninitialized`：

```
$gcc -O -Wuninitialized blunder.c -o blunder
blunder.c: In function 'main':
blunder.c:9: warning 'j' might be used uninitialized in this function
blunder.c: At top level:
blunder.c:27: warning: type mismatch with previous implicit
declaration
blunder.c:21: warning: previous implicit declaration of 'no_decl'
blunder.c:27: warning: 'no_decl' was previously implicitly declared
to return 'int'
```

有趣的是，GCC 没有警告变量 `i` 未初始化就使用的情况，虽然它指出了变量 `j` 的问题。这是因为在变量 `i` 上首先标记了一个 `-Wparentheses` 警告（你可以通过一起使用 `-Wparentheses` 和 `-Wuninitialized` 选项来证实这一点）。如果要抓出所有这些警告，甚至比这更多，可以使

用后面介绍的-Wall 选项。采用这个选项可以大大缩短输入的命令行长度。

注意： 最后一个例子中用到了-O (优化) 选项。这样做是必要的，因为即使没有在 GCC 的 info 页面中说明，可是-Wuninitialized 选项确实要求使用-O 选项。

本节展示了 GCC 抓出真实的和潜在的编程错误的能力。下一节介绍 GCC 的另一种功能：代码优化。

3.4 优化选项

代码优化的目的是改进程序的执行效率。其代价是延长了编译的时间、增加了编译时内存的用量，而且有时候最终生成的二进制文件会占用更多的硬盘空间。有的优化本质上很普通，能够在任何情况下应用，另外一些优化则专门用于发挥某种 CPU 或 CPU 系列的特性。这两类优化本节都会介绍。

仅使用-O 选项告诉 GCC 同时减小代码长度和执行时间。这个选项等价于-O1。在这个级别上能够执行的优化类型取决于目标处理器，但一定会包含线程直接跳转(thread jump)和延迟退栈(deferred stack pops)这两种优化。线程直接跳转优化的目的是减少跳转操作的次数；延迟退栈则通过在嵌套函数调用时推迟退栈的时间而优化运行效率，未优化的情况下每次函数调用完成后返回时都需要弹出栈中的函数参数，而优化后在栈中保留了参数，直到完成所有递归调用后才同时弹出栈中积累的函数参数。

-O2 级的优化包括所有-O1 级的优化和额外一些调整，其中包含对处理器指令调度的调整。在这个级别上，编译器保证处理器在等待其他指令的结果或者在等待来自二级高速缓存或主内存的数据时，仍然有可执行的指令。但是这种优化的实现和具体的 CPU 类型高度相关。-O3 选项包括所有的-O2 级优化，还包括循环展开和其他与处理器特性有关的优化。

根据你对某种 CPU 系列底层知识了解的多少，可以使用-f{flag}选项来请求需要执行的特定优化。表 3.4 列出了通常比较有用的 8 种-f 优化标志。

表 3.4 GCC 优化标志

选项	作用
-ffloat-store	禁止在 CPU 的寄存器中保存浮点变量的值。这能把 CPU 寄存器节省下来留作它用，而且可以防止产生过分精确但不必要的浮点数
-ffast-math	产生浮点数学优化，这能提高速度但违反了 IEEE 或 ANSI/ISO 标准。如果程序不需要严格遵守 IEEE 规范，可在编译浮点密集型的程序时考虑采用这一标志
-finline-functions	把所有的“简单”函数在调用它们的函数中就地展开。编译器决定了什么是“简单”函数。减少处理器与函数相关的开销是一种基本的优化技术
-funroll-loops	展开所有能在编译时确定重复次数的循环体。展开循环体后每步循环都能省出几条 CPU 指令，这样大大减少了执行时间
-fomit-frame-pointer	如果函数不需要则丢掉帧指针，该指针保存在 CPU 的一个寄存器中。因为去掉了设置、保存和恢复帧指针所必需的指令，所以加快了处理速度

(续表)

选项	作用
-fschedule-insns	记录可能暂停的指令，因为它们正在等候的数据不在 CPU 中
-fschedule-insns2	执行第二次指令重排序(类似于-fschedule-insns)
-fmove-all-movables	把所有出现在循环体内部但稳定不变的计算移出循环体。这从循环体中去除了不必要的操作，加快了循环的整体运行速度

内联和循环展开技术都能够大大提高程序的执行速度，因为它们都避免了函数调用和变量查找的开销，但付出的代价往往是大大增加了目标或二进制代码的大小。需要通过试验来确定，相对于更大的结果文件来说，所取得更快的执行时间或其他优化结果是否值得。总体而言，当使用表 3.4 列出的编译选项时，有必要进行试验和代码剖析(profiling)，或者性能分析来证实某种优化措施确实达到了预期的效果。

作为试验，下面的程序 `pisqrt.c` (参见程序清单 3.7) 计算了 10 000 000 次 π 的平方根。表 3.5 列出了编译这个程序所使用的优化或处理器标志选项，以及在一台具有 Pentium II 266MHz CPU 和 128MB 内存的主机上 10 次执行 `pisqrt` 的平均时间。

程序清单 3.7 计算 π 的平方根

```

/*
 * pisqrt.c - Calculate the square of PI 100,000,000
 * times
 */
#include <stdio.h>
#include <math.h>

int main(void)
{
    double pi = M_PI;    /* Defined in <math.h> */
    double pisqrt;
    long i;

    for (i = 0; i < 10000000; ++i) {
        pisqrt = sqrt(pi);
    }
    return 0;
}

```

表 3.5 `pisqrt` 的执行时间

标志/优化	平均执行时间
<none>	5.43s
-O1	2.74s
-O2	2.83s
-O3	2.76s
-ffloat-store	5.41s

(续表)

标志/优化	平均执行时间
-ffast-math	5.46s
-funroll-loops	5.44s
-fschedule-insns	5.45s
-fschedule-insns2	5.44s

这个非常不严格的试验表明,至少对于这个程序来说,最大的性能改进是通过使用-O1、-O2 或-O3 选项由编译器选择合适的优化功能集合得到的。这个例子说明,除非对处理器的体系结构非常了解或者知道某种特殊的优化专门针对你的程序有影响,否则就应该使用优化选项-O。

提示: 一般而言, Linux 程序员似乎爱用优化选项-O2。即使对于像本章开头介绍的 hello.c 那样很小的程序,使用这个选项后,在执行文件大小和执行时间上也可以发现有细微的改进。但 Linux 程序员使用这个选项更多是出于习惯而不是测试经验。正如表 3.5 所显示的那样,对于被测程序来说,优化选项-O1 得到了最大的性能提高。这说明了什么? 尝试不同的优化级别,看看哪个是最好的结果!

3.5 调 试 选 项

错误是不可避免的,就如同死亡和缴税是不可避免的一样。为了面对这一无法回避的现实,可以使用 GCC 的-g 和-ggdb 选项在编译后的程序中插入调试信息以方便调试会话过程。另外, GCC 还有一些选项能让代码剖析会话过程更简单、效率更高。

能够用 1、2 或 3 来限定-g 选项来指定产生多少调试信息。默认的级别是 2(-g2),此时产生的调试信息包括扩展的符号表、行号以及局部或外部变量的信息。这些信息全部保存在二进制文件里。3 级调试信息包括所有的 2 级信息和源代码中定义的所有宏。相反,1 级产生的信息只够创建回溯(backtrace)和堆栈转储(stack dump)之用。回溯是指一个程序调用函数的历史。堆栈转储是一个通常以原始的十六进制格式保存程序执行环境内容的列表,列表内容主要是 CPU 寄存器和分配给程序的内存。注意,1 级调试不产生局部变量和行号的调试信息。

如果你打算使用 GNU Debugger,即 gdb(在第 8 章“调试”中介绍),可以用-ggdb 选项来创建额外的调试信息以方便采用 gdb 进行的调试工作。但是,这样做也可能会导致程序不能用其他调试器(比如 Solaris 操作系统上常用的 DBX)进行调试。-ggdb 能接受的调试级别规范和-g 的一样,它们对调试输出有相同的影响。

但是,使用任何一种启动调试的选项都会让二进制文件的大小急剧增长。在笔者的系统上编译和链接本章前面介绍的那个简单的 hello.c 程序产生的二进制文件只有 4089 字节大小。当我用-g 和-ggdb 选项编译它时产生的结果之大可能会让你感到惊异:

```
$ gcc -g hello.c -o hello
$ ls -l hello
-rwxr-xr-x 1 kwall users 10275 May 21 23:27 hello

$ gcc -ggdb hello.c -o hello
$ ls -l hello
-rwxr-xr-x 1 kwall users 8135 May 21 23:28 hello
```

读者可以看到，`-g` 选项让二进制文件大小增长到将近三倍，而 `-ggdb` 选项也让其大小增加了一倍！尽管会使文件大小增长，笔者仍然建议在执行文件中包含标准的调试符号（使用 `-g` 选项创建），以便某些用户在遇到问题时可以调试你的代码。

其他的调试选项包括 `-p` 和 `-pg`，它们将剖析（profiling）信息加入二进制文件中。这些信息对于找出代码中的性能瓶颈以及开发高性能的程序非常有帮助。`-p` 选项在代码中加入 `prof` 程序能够读取的剖析符号信息，而 `-pg` 选项加入了 GNU 项目中 `prof` 的化身 `gprof` 能够解释的符号信息。`-a` 选项在代码中加入了代码块（比如函数）累计使用的次数。

`-save-temps` 选项可以保存在编译过程中生成的中间文件，其中包括目标文件和汇编代码文件。如果你怀疑编译器对你的代码执行了不正常的操作，或者你想检查生成的代码了解是否能够通过手工调整提高性能，那么这些文件会给你提供帮助。

如果你对编译器到底花费了多少时间来完成它的工作感兴趣，可以考虑使用 `-Q` 选项，这个选项让 GCC 显示编译过程中碰到的每个函数，并提供编译器编译每个函数所花时间的剖析信息。例如，编译 `hello.c` 程序时的输出结果如下：

```
$ gcc hello.c
main
time in parse: 0.020000
time in integration: 0.000000
time in jump: 0.000000
time in cse: 0.000000
time in loop: 0.000000
time in cse2: 0.000000
time in branch-prob: 0.000000
time in flow: 0.000000
time in combine: 0.000000
time in regmove: 0.000000
time in sched: 0.000000
time in local-alloc: 0.000000
time in global-alloc: 0.000000
time in sched2: 0.000000
time in shorten-branch: 0.000000
time in stack-reg: 0.000000
time in final: 0.000000
time in varconst: 0.000000
time in symout: 0.000000
time in dumpt: 0.000000
```

显示的时间可能随系统的不同而不同。编译器的编写人员对这种信息最感兴趣，但是

如果对编译器正在干什么感到很好奇，那么现在你就会知道怎样去得到答案了。

最后，如笔者在本章开始时所说，GCC 允许在优化代码的同时插入调试信息。优化代码对于调试而言是一个挑战，因为在优化以后，在源程序中所声明和使用的变量很可能不再使用，控制流也可能会突然跳转到意外的地方，计算常量的语句很可能不被执行，而且由于循环被展开，循环中的语句也会到处出现，等等。因此，笔者个人的偏好是在优化以前彻底调试程序，也许读者的选择会有所不同。

提示： 不要因为编译器的优化能力而忽略在程序设计阶段的优化工作。在本章中提及的优化只是指编译器所能做的那部分优化。但是，良好的设计和高效的算法对程序执行效率的影响远比编译器优化的影响大得多。例如，除了在非常小的数据集下，高度优化的冒泡排序也不会和快速排序一样快。如果有了简洁的设计和快速算法，那么可能就不需要编译器来优化代码了，即便这样做没有什么坏处。

3.6 特定体系结构的选项

除了上一节讨论的优化选项之外，GCC 还可以生成专门针对每种类型的 CPU 的代码。要做到这一点，需使用 `-m{value}` 选项。表 3.6 给出了针对 Intel i386 系列处理器的选项。

表 3.6 特定体系结构的 GCC 选项

选项	含义
<code>-mcpu=CPU TYPE</code>	编译时使用针对 CPU_TYPE 的默认 CPU 指令调度。CPU TYPE 可以选择 i386、i486、i586、pentium、i686 以及 pentiumpro
<code>-m386</code>	和 <code>-mcpu=i386</code> 同义
<code>-m486</code>	和 <code>-mcpu=i486</code> 同义
<code>-mpentium</code>	和 <code>-mcpu=pentium</code> 同义
<code>-mpentiumpro</code>	和 <code>-mcpu=pentiumpro</code> 同义
<code>-march=CPU TYPE</code>	针对 CPU TYPE 生成指令。CPU TYPE 可以选择 i386、i486、i586、pentium、i686 以及 pentiumpro。-march=CPU TYPE 隐含了 <code>-mcpu=CPU TYPE</code>
<code>-mieee-fp</code>	浮点数比较时使用 IEEE 标准
<code>-mno-ieee-fp</code>	浮点数比较时不使用 IEEE 标准
<code>-malign-double</code>	将 double、long double 和 long long 变量按照双字（double word）对齐，这样生成的代码速度更快
<code>-mno-align-double</code>	不把 double、long double 和 long long 变量按照双字对齐
<code>-mrtld</code>	强行将参数个数固定的函数用 <code>ret NUM</code> 指令返回，节省调用函数的一条指令

有必要对这些选项作几点说明。通过在 `-mcpu=CPU_TYPE` 中指定某种 CPU 类型能够生成适合于特定 CPU 的指令，但同时必须使用 `-march=CPU_TYPE` 选项，否则编译器生成的代码也能运行在 i386 处理器上。类似地，要针对某种给定的处理器生成代码以及进行指

令调度，也要使用 `-march=CPU_TYPE` 选项。这是针对某种给定的处理器定制编译代码的最好办法。如表 3.6 所指出的那样，`-malign_double` 选项能够产生稍快一点的代码，但它只对 Pentium 处理器起作用。`-mrtld` 选项覆盖了前面讨论的 `-fdefer-stack-pops` 选项。

3.7 GNU C 扩展

GNU CC (GNU Compiler Collection) 在许多方面对 ANSI 标准进行了扩展。如果不介意直接编写非标准的代码，那么其中的一些扩展可能会带来方便且非常有用。

3.7.1 关于可移植性

但是非标准代码带来的问题是它的可移植性不好——毫无疑问，利用了 GNU 扩展特性的代码不能用非 GNU 的编译器编译。C 语言始终能够保持良好的适应性和稳定性的原因之一，除了它固有的强大功能和灵活性外，还因为它有标准，不但移植到了现有的每一种主要的计算机体系结构上，还可以移植到许多次要的平台之上。标准化的工作使它极易移植。

注意：GCC 也对 C++ 语言进行了扩展。

所以说，勇敢的读者必须在扩展的方便性和编写兼容 ANSI/ISO 标准的 C 代码之间进行抉择。我建议编写标准 C 代码。当偏离标准之后，使用 POSIX 函数（比如读写文件描述符，这将在第 11 章和第 12 章中介绍）就会发生这个问题，必须要把非标准代码放入一个独立的模块并采取措施以保证程序在严格的 ANSI/ISO 系统中也能编译。达到上述目的通常做法是用编译预处理指令 `#ifdef` 把非标准的代码括起来。下面的代码段演示了这种方法。它既可以在严格遵循 ANSI C 标准的环境下编译，也能在相对宽松的 GNU 环境下编译：

```
#ifdef __STRICT_ANSI__
/* use ANSI/ISO C only here */
#else
/* use GNU extensions here */
#endif
```

如果用户或是 ANSI 兼容的编译器定义了 `__STRICT_ANSI__` 宏，则表明需施加 ANSI 兼容的环境，并编译 `#ifdef` 语句块的第一部分代码。否则，编译 `#else` 后面的代码。

对于所有的细节，我建议好奇的读者阅读 GCC 的 Info 页面。本节介绍的是在 Linux 系统的头文件、源代码文件以及许多 Linux 应用程序中常见的扩展。

注意：ANSI/ISO C 标准既没有定义编译器的行为，也没有定义当源代码使用某种结构（比如返回 void 的 main 函数）时编译器应该生成的程序。至少从理论上说，任何事情都可能发生。新闻组 `comp.lang.c` 上的一个老说法是，如果你调用了未定义的行为，那么守护进程会从你的鼻孔里飞出去。引用这个说法的目的是为了说明，一个程序不正确或不能移植的原因在于它是在你的系统上用你的编译器编译的。

3.7.2 GNU 扩展

例如, GCC 使用 `long long` 类型来提供 64 位存储单元:

```
long long long_int_var;
```

在 x86 平台上, 这一声明为 `long_int_var` 分配了 64 位的存储空间。

注意: 在新的 ISO C 标准草案中包含了 `long long` 类型。

内联函数

在 Linux 头文件中遇到的另一类 GCC 扩展就是内联函数的使用。如果足够短小, 内联函数就能像宏一样在代码中展开, 这样就减少函数调用的开销; 同时, 因为编译器在编译时能够对内联函数进行类型检查, 所以使用内联函数要比宏安全。

要使用内联函数, 需在函数的返回类型前插入关键字 `inline`, 如下面的代码片段所示, 还要在编译时使用 `-O` 优化选项。

```
inline void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

这个短小的函数以内联函数的方式实现了人们熟知的交换函数。但是请注意, 对于编译器来说关键字 `inline` 只是一个建议而不是指令。编译器通过其内部的判别机制来决定一个函数能否或者是否作为内联函数编译。

函数和变量属性

关键字 `attribute` 通过向 GCC 指明有关代码的更多信息来帮助代码优化工作进行得更好。例如, 标准库函数 `exit` 和 `abort` 都不返回调用它们的函数。编译器如果知道它们不返回就能生成效率稍高的代码。当然, 用户程序也能定义不返回的函数。GCC 允许为这些函数指定 `noreturn` 属性, 作为编译器在优化该函数时的提示。

例如, 假定有个没有返回的函数 `die_on_error`。为了使用函数属性, 可以在函数声明后面加上 `__attribute__((attribute_name))`。于是函数 `die_on_error` 的声明如下:

```
void die_on_error(void) __attribute__((noreturn));
```

函数还和平常一样来定义:

```
#include<stdlib.h>
void die_on_error(void)
{
    /* your code here */
    exit(EXIT_FAILURE);
}
```

也可以对变量指定属性。例如，`aligned` 属性指示编译器在为变量分配内存空间时按指定字节数对齐边界。下列语句：

```
int int_var __attribute__((aligned 16)) = 0;
```

使 GCC 让变量 `int_var` 的边界按 16 字节对齐。`packed` 属性告诉 GCC 为变量或结构分配最小的内存空间。在和结构类型一起使用时，`packed` 属性使得 GCC 不再像往常那样为对齐左边界而插入额外的填充字节。

如果想要关闭对未用变量发出的所有警告，那么可以对变量使用 `unused` 属性，它告诉编译器该变量不准备使用。下面的变量声明会消除警告：

```
float big_salary __attribute__((unused));
```

注释定界符

除非使用 `-ansi` 或 `-traditional` 编译选项，否则 GCC 允许在 C 语言中使用 C++ 的注释定界符 `//`。许多其他编译器也允许这样，而这可能会成为当前正在修订的 C 新标准的内容。这个特性对于那些从广泛使用 C++ 而不是 C 语言的大学中走出来的新一代程序员，以及更习惯于输入 `//` 而不是 `/*` 和 `*/` 的人来说，确实方便不少。把这种扩展看作是一种方便吧。

使用 case 区间

`case` 区间是一个非常有用的扩展。其语法如下：

```
case LOWVAL ... HIVAL:
```

注意，在省略号前后必须有空格。在 `switch` 语句中，`case` 区间指定了落在 `LOWVAL` 和 `HIVAL` 区间内的那些整数值。例如：

```
switch(int_var) {
case 0 ... 2:
    /* your code here */
    break;
case 3 ... 5:
    /* more code here */
    break;
default:
    /* default code here */
}
```

上面的代码片段等价于：

```
switch(int_var) {
case 0:
case 1:
case 2:
    /* your code here */
    break;
case 3:
```

```
case 4:
case 5:
    /* more code here */
    break;
default:
    /* default code here */
}
```

case 区间只是对于 switch 语句传统语法的简略记法而已。正如你所看到的那样，它不但使得第一段代码更简短，而且还稍稍提高了它的可读性（虽然老道的 C 程序员认为符合惯用语法的第二段代码更容易阅读）。

构造函数名称

把函数名用作字符串是 GNU 的扩展，它能极大地简化调试工作。GCC 预先定义了变量 `__FUNCTION__` 为当前函数（控制流程当前所在的位置）的名字，就好象它被写到源代码里去了一样。程序清单 3.8 演示了这一特性是如何起作用的。

程序清单 3.8 使用 `__FUNCTION__` 变量

```
/*
 * showit.c - Illustrate using the __FUNCTION__ variable
 */

#include <stdio.h>

void foo(void);

int main(void)
{
    printf("The current function is %s\n", __FUNCTION__);
    foo();
    return 0;
}

void foo (void)
{
    printf("The current function is %s\n", __FUNCTION__);
}
```

这个程序的输出如下：

```
$/showit
The current function is main
The current function is foo
```

正如你所看到的那样，showit 善意地输出了当前正在执行的函数的名字。如果在调试期间难以确定导致程序出问题的地方，那么这个功能就很有用了。只要插入几条使用 `__FUNCTION__` 变量的 printf 语句就能很快缩小出现问题的范围，直至最终找到故障点。

3.8 pgcc: 奔腾处理器的编译器

在结束本章之前, 值得提提另一种知名度不如 GCC 的编译器 pgcc, 即 Pentium GCC。pgcc 是由 Pentium 编译器小组(Pentium Compiler Group, <http://www.goof.com/pcg/>)维护的, 在 GCC 针对 Pentium 的优化做得不好时, 人们创建了 pgcc 来处理针对 Pentium 处理器体系结构的不同的优化特性。虽然 pgcc 代表了 GCC 基本代码的一个分支, 但其维护者还是紧紧跟随着 GCC 的版本。实际上, pgcc 是作为 egcs 的一组补丁发布的, 后者是目前正式的 GNU 编译器。

使用 pgcc 的主要好处是它对 Pentium 处理器的优化较好。一组 Intel 工程师最初是在某个 GCC 版本的基础上为 Pentium 处理器创建了 pgcc。虽然 Intel 的小组公布的评测数据显示在某些应用上 pgcc 会提高性能 30%, 但是 Pentium 编译器小组却提醒人们, 在现实环境下 pgcc 可能只会提高 5% 的性能。

为什么会有 pgcc 呢? 特别是在目前 egcs 已经融入了许多高级的 Pentium 优化选项的情况下似乎没有必要。首先你不需要。但别忘了, 这是 Linux, 任何人可以自由地做自己觉得合适的事情。既然 Linux 的两个发布版本 Stampede 和 Enoch 都采用 pgcc 作为编译器, 所以它还是有些优点的。它还代表了一种替代 GCC 的选择。再进一步说, 看看 pgcc 是否能在你的系统上产生更快或更小的二进制文件也可作为一个有趣的试验。最后, 用用别的软件也挺好玩嘛。

3.9 小 结

本章向读者介绍 GCC(GNU Compiler Collection)。在一个简短的示例程序之后, 本章讲解了许多 GCC 的特性, 包括使用库文件和头文件的选项, 生成编译时的警告信息, 向程序加入调试信息以及优化。实际上, 这些内容只是些皮毛; GCC 自己的文档有数百页之多。然而, 了解 GCC 的特性和功能有助于读者开始在自己的开发项目中使用它们。有了这些基本技能, 你就可以准备开始编程了。但是还是先阅读下一章“使用 GNU make 管理项目”, 它向你的 Linux 编程工具箱中增添了另一个关键的工具。

第 4 章 使用 GNU make 管理项目

在本章中读者会看到有关 `make` 的介绍，`make` 是一种控制编译或者重复编译软件的工具。`make` 可以自动管理软件编译的内容、方式和时机，从而使程序员能够把精力集中在编写代码上。

4.1 为何使用 `make`

除了最简单的软件项目，`make` 对于其他所有项目而言都很必要。首先，包含多个源代码文件的项目在编译时都有长而且复杂的命令行。而且，编程项目经常需要使用那些很少用到且难以记忆的特殊编译选项。`make` 可以通过把这些复杂而难记的命令行保存在 `makefile` 文件中来解决上述两个问题，`makefile` 将在下一小节讨论。

`make` 还能减少重复编译所需要的时间，因为它很聪明，能够判断哪些文件被修改过，进而只重新编译程序被修改过的部分。`makefile` 为项目构建了一个依赖信息数据库，因而可以让 `make` 在每次编译前检查是否可以找到所有需要的文件。`make` 还可以让你建立一个稳定的编译环境。最后，`make` 可以让编译过程自动执行，因为从 `shell` 脚本或者 `cron`（定时）作业调用 `make` 非常容易。

4.2 编写 `makefile`

`make` 是怎样完成这些神奇工作的呢？是通过使用 `makefile` 文件做到的。`makefile` 是一个文本形式的数据库文件，其中包含一些规则告诉 `make` 编译哪些文件、怎样编译以及在什么条件下去编译。每条规则包含以下内容：

- 一个“目标体”（`target`），即 `make` 最终需要创建的东西。
- 包含一个或多个“依赖体”（`dependency`）的列表，依赖体通常是编译目标体需要的其他文件。
- 为了从指定的依赖体创建出目标体所需执行的“命令”（`command`）的列表。

虽然目标体通常都是程序，但它们可以是诸如文本文件、手册页面等任何东西。目标体甚至能测试和设置环境变量。类似地，也可以定义依赖体以确保编译开始前存在某个特殊的环境变量。最后，`makefile` 中的命令可以是编译器的命令或 `shell` 命令，它们能设置环境变量、删除文件，或者任何命令行所能完成的功能，如从 `FTP` 站点下载文件等。`GNU make` 被调用后会顺序查找名为 `GNUmakefile`、`makefile` 或 `Makefile` 的文件。出于某种原因，可能只是习惯和长期形成的约定吧，大多数 `Linux` 程序员使用最后一种形式 `Makefile`。

Makefile 规则有下列通用形式:

```
target : dependency [dependency [...]]
        command
        command
        [...]
```

警告: 每一个命令的第一个字符必须是制表符, 仅使用 8 个空格是不够的。这一点经常不被人们注意, 并且当所使用的编辑器友好的将制表符转换成 8 个空格时, 会产生问题; 因为如果用空格代替制表符, make 会在执行过程中显示 Missing Separator (缺少分隔符) 并停止。

target 是需要创建的二进制文件或目标文件。dependency 是在创建 target 时需要输入的一个或多个文件的列表。命令序列是创建 target 文件所需要的步骤, 如编译命令。此外, 除非特别指定, 否则 make 的工作目录就是当前目录。

4.3 编写 makefile 的规则

如果上一节的内容对你来说太抽象, 那么本节使用程序清单 4.1 再具体讨论。这是用于编译第 3 章中出现的程序 howdy 和 hello 的 makefile 文件。

程序清单 4.1 演示目标体、依赖体和命令的简单 makefile 文件

```
howdy: howdy.o helper.o helper.h
    gcc howdy.o helper.o -o howdy

helper.o: helper.c helper.h
    gcc -c helper.c

howdy.o: howdy.c
    gcc -c howdy.c

hello:hello.c
    gcc hello.c -o hello

all: howdy hello

clean:
    rm howdy hello *.o
```

要编译 howdy, 只需在 makefile 所在目录下输入 make 即可。就这么简单。

这个 makefile 文件包含 6 条规则。第一个目标体 howdy 称为默认 (default) 目标体——这是 make 要创建的文件。howdy 有 3 个依赖体, 分别为 howdy.o、helper.o 和 helper.h; 要编译生成 howdy, 必须要有这 3 个文件。第二行调用编译器的命令供 make 执行来创建 howdy。由对第 3 章内容的回忆可知, 这条命令从两个目标文件创建名为 howdy 的可执行文件。把头文件 helper.h 作为一个依赖体列入是为了避免编译器调用未声明的函数产生出错信息。接下来的两条规则告诉 make 怎样生成单个目标文件, helper.o 和 howdy.o。这些

规则使用了 gcc 的 `-c` 选项，只创建目标文件但跳过链接。如果只想生成两个目标文件而不生成 `howdy` 本身，可以使用下面两条命令：

```
$ make helper.o
$ make howdy.o
```

更简洁一点，只需使用

```
$ make helper.o howdy.o
```

正如你所看到的那样，`make` 允许把多个目标作为参数。这两种方法都能使用相应的规则和命令生成目标文件。图 4.1 给出了这个过程的图示。

图 4.1 把生成 `howdy` 的步骤归结到第 3 章讨论的一般性的预处理/编译/链接过程上。`howdy.c` 和 `helper.c` 这两个源代码文件经预处理后编译成目标文件。然后链接器把来自文件 `howdy.o` 和 `helper.o` 的目标代码和标准库以及 C 启动代码链接到一起生成二进制文件 `hello`。

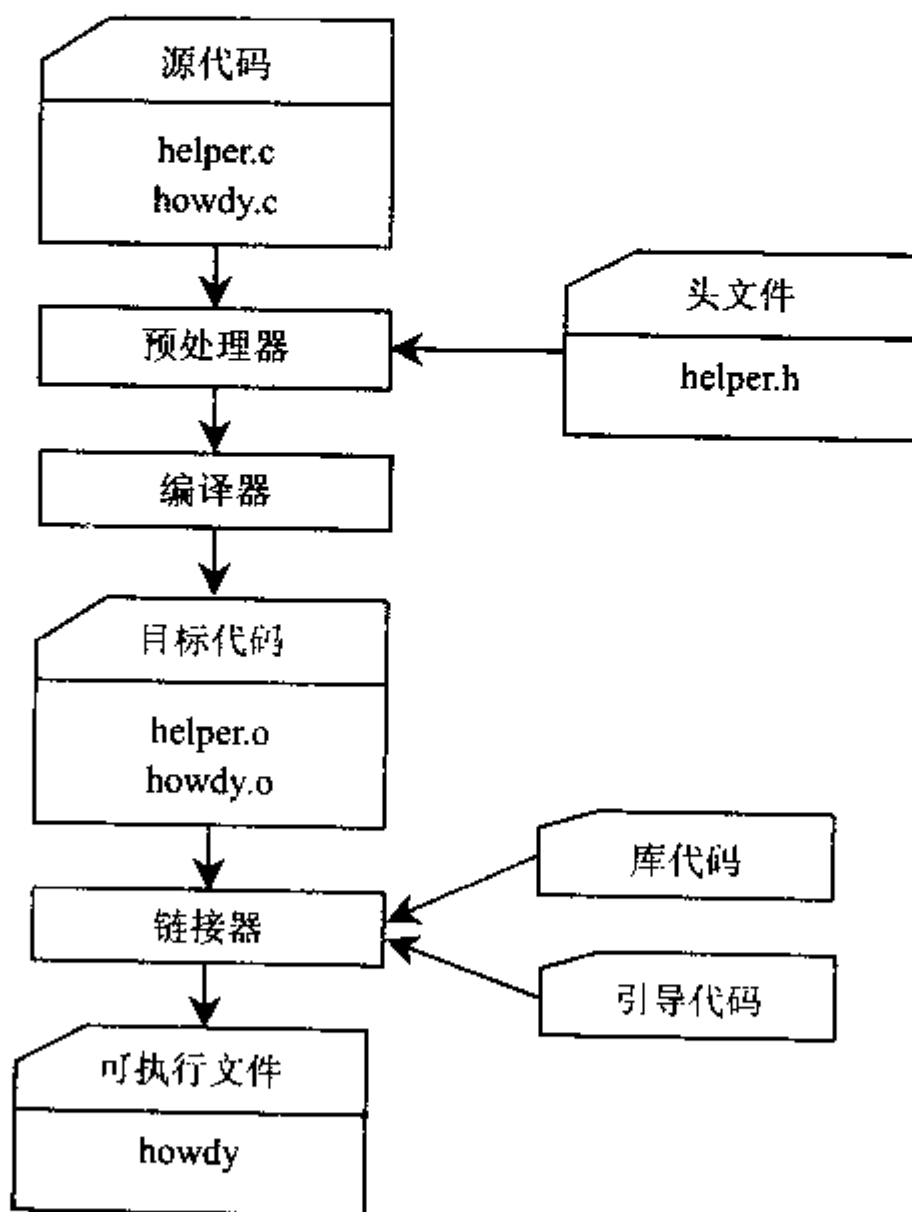


图 4.1 从一个 makefile 文件创建 `howdy`

现在，`make` 的价值就体现出来了：通常情况下，如果试图在依赖体 `helper.o` 和 `howdy.o` 不存在的情况下使用所示的命令编译 `howdy`，则 `gcc` 会报错并退出。另一方面，在看到 `howdy` 需要这两个文件（和 `helper.h`）后，`make` 先确认它们是否存在，如果不存在则首先执行命令生成它们，然后再返回到第一条规则创建可执行文件 `howdy`。当然，如果 `helper.h` 不存在，`make` 也会放弃执行，因为它没有创建 `helper.h` 的规则。

“一切都很好”，也许你会这么想，“但是 make 怎样知道什么时候需要重新编译一个文件呢？”答案极其简单：如果指定的目标文件 make 找不到，make 就会生成它。如果目标体存在，make 会对目标体文件和依赖体文件的时间戳进行比较。如果有一个或多个依赖体比目标体新，make 就重新编译生成目标体，因为 make 认为新的依赖体意味着对代码做过修改，必须把改动融入到目标体中去。

第四条规则相当简单。它定义了如何编译生成第3章介绍的简单程序 hello。第五条是创建 hello 和 howdy 的笼统规则，它还表明甚至二进制文件都能作依赖体。下一小节将讨论第六条规则，即伪目标。

4.3.1 伪目标

除了一般的文件目标体，比如 howdy 和 hello 之外，make 也允许指定伪目标。称其为伪目标是因为它们并不对应于实际的文件。程序清单 4.1 中最后一个目标体 clean 就是伪目标。伪目标体规定了 make 应该执行的命令。但是，因为 clean 没有依赖体，所以它的命令不会被自动执行。下面解释 make 是如何工作的：当遇到目标体 clean 时，make 先查看其是否有依赖体，因为 clean 没有依赖体，所以 make 认为目标体是最新的而不执行任何操作。为了编译这个目标体，必须输入 make clean。在本例中，clean 删除了可执行文件 hello 和 howdy 以及构成 howdy 的目标文件。在创建和发行仅包含源代码的压缩包或者需要彻底重新编译时可能会用到这样一个目标体。

然而，如果恰巧有一个名为 clean 的文件存在，make 就会发现它。然后和前面一样，因为 clean 没有依赖体文件，make 就认为这个文件是最新的而不会执行相关命令。为了处理这类情况，需要使用特殊的 make 目标体 .PHONY。 .PHONY 的依赖体文件的含义和通常一样，但是 make 不检查是否存在有文件名和依赖体中的一个名字相匹配的文件，而是直接执行与之相关的命令。在使用了 .PHONY 之后，前面的例子如下：

程序清单 4.2 带有 PHONY 目标的 Makefile 文件

```
howdy: howdy.o helper.o helper.h
    gcc howdy.o helper.o -o howdy

helper.o: helper.c helper.h
    gcc -c helper.c

howdy.o: howdy.c
    gcc -c howdy.c

hello: hello.c
    gcc hello.c -o hello

all: howdy hello

.PHONY : clean

clean:
    rm howdy hello *.o
```

4.3.2 变量

为了简化编辑和维护 `makefile`, `make` 允许在 `makefile` 中创建和使用变量。所谓的变量其实是用指定文本串在 `makefile` 中定义的一个名字, 这个文本串就是变量的值。下面是定义变量的一般方法:

```
VARNAME=some_text [...]
```

把变量用括号括起来, 并在前面加上“\$”符号, 就可以引用变量的值:

```
$(VARNAME)
```

此时, `VARNAME` 在等式右端展开为它所代表的文本。变量一般都在 `makefile` 的头部定义, 并且, 按照惯例, 所有的 `makefile` 变量都应该是大写 (虽然这不是必须的)。这样, 如果变量的值发生变化, 就只需要在一个地方修改, 从而简化了 `makefile` 的维护。现在, 继续现在修改程序清单 4.1, 加入两个变量, 结果如程序清单 4.3 所示。

程序清单 4.3 在 `makefile` 中使用变量

```
OBJS = howdy.o helper.o
HDRS = helper.h
howdy: $(OBJS) $(HDRS)
    gcc $(OBJS) -o howdy

helper.o: helper.c $(HDRS)
    gcc -c helper.c

howdy.o: howdy.c
    gcc -c howdy.c

hello: hello.c
    gcc hello.c -o hello

all: howdy hello

clean:
    rm howdy hello *.o
```

`OBJS` 和 `HDRS` 在被引用的每个地方都展开成它的取值。编译时也是如此。

实际上, `make` 使用两种变量: 递归展开变量和简单展开变量。递归展开变量在引用时逐层展开, 即如果在展开式中包含了对其他变量的引用, 则这些变量也将被展开, 直到没有需要展开的变量为止, 这就是所谓的递归展开。下面的例子有助于弄清这个概念。

假设变量 `TOPDIR` 和 `SRCDIR` 如下定义:

```
TOPDIR = /home/kwall/myproject
SRCDIR = $(TOPDIR)/src
```

这样, `SRCDIR` 的值是 `/home/kwall/myproject/src`, 则工作正常。但是, 考虑下面的变量定义:

```
CC = gcc
CC = $(CC) -o
```

很清楚，定义者想要得到的结果是“CC=gcc -o”。但是实际并非如此；CC 在被引用时递归展开，从而陷入一个无限循环中：CC 将扩展为\$(CC) 的值，从而永远也读不到-o 选项。幸运的是，make 能够检测到这个问题并报告错误：

```
*** Recursive variable 'CC' references itself(eventually). Stop
```

为了避免这个问题，可以使用简单展开变量。与递归展开变量在引用时展开不同，简单展开变量在定义处展开，并且只展开一次，从而消除了变量的嵌套引用。在定义时，其语法与递归展开变量有细微的不同：

```
CC := gcc -o
CC += -O2
```

第一个定义使用“:=”设置 CC 的值为 gcc -o，第二个定义使用“+=”在前面定义的 CC 后附加了-O2，从而 CC 最终的值是 gcc -o -O2。如果在使用 make 变量时遇到“VARNAME references itself”这类错误信息，就可以使用简单展开变量来解决。一些程序员仅使用简单展开变量，以避免出现意想不到的问题；但既然现在是在 Linux 上，你可以自由选择使用的方式。

除用户定义变量外，make 也允许使用环境变量、自动变量和预定义变量。使用环境变量非常简单。在启动时，make 读取已定义的环境变量，并且创建与之同名同值的变量。但是，如果 makefile 中有同名的变量，则这个变量将取代与之相应的环境变量，所以应当注意这一点。

此外，make 也提供一长串预定义变量和自动变量，但是它们看起来有些神秘。之所以称为自动变量是因为 make 自动用特定的、熟知的值替换它们。表 4.1 给出了部分自动变量。

表 4.1 自动变量

变量	说明
\$@	规则的目标所对应的文件名
\$<	规则中的第一个相关文件名
\$^	规则中所有相关文件的列表，以空格为分隔符
\$?	规则中日期新于目标的所有相关文件的列表，以空格为分隔符
\$(@D)	目标文件的目录部分（如果目标在子目录中）
\$(@F)	目标文件的文件名部分（如果目标在子目录中）

除了表 4.1 列出的自动变量外，make 还预定义了许多其他变量，用于定义程序名或给这些程序传递标志和参数。这些预定义的变量看上去更像常规的 make 变量而不是像字符名称的自动变量。表 4.2 给出了一些有用的预定义变量。

表 4.2 用于程序名和标志的预定义变量

变量	说明
AR	归档维护程序，默认值=ar
AS	汇编程序，默认值=as

(续表)

变量	说明
CC	C 编译程序, 默认值=cc
CPP	C 预处理程序, 默认值=cpp
RM	文件删除程序, 默认值="rm -f"
ARFLAGS	传给归档维护程序的标志, 默认值=rv
ASFLAGS	传给汇编程序的标志, 没有默认值
CFLAGS	传给 C 编译器的标志, 没有默认值
CPPFLAGS	传给 C 预处理程序的标志, 没有默认值
LDFLAGS	传给链接程序 (ld) 的标志, 没有默认值

如果需要, 可以在 `makefile` 中重新定义这些变量, 但是在大多数情况下, 这些默认值都是合理的。

4.3.3 隐式规则

除了在 `makefile` 文件中显式指定的规则 (称为显式规则) 外, `make` 还有一整套隐式规则, 或称为预定义规则。这些规则多数有特殊目的而且用途有限, 所以在这里只介绍几种最常用的隐式规则。隐式规则简化了 `makefile` 的编写和维护。

假设有下面这样的一个 `makefile`:

```
OBJS = editor.o screen.o keyboard.o
editor : $(OBJS)
    cc -o editor $(OBJS)

.PHONY : clean

clean :
    rm editor $(OBJS)
```

默认目标 `editor` 所对应的命令提及了 `editor.o`、`screen.o` 和 `keyboard.o`, 但是 `makefile` 中没有怎样编译生成这些目标的规则。此时, `make` 就使用所谓的隐式规则, 实际上, 对每一个名为 `somefile.o` 的目标 (object) 文件, `make` 首先找到与之相应的源代码 `somefile.c`, 并且用 `gcc -c somefile.c -o somefile.o` 编译生成这个目标文件。所以, 在本例中 `make` 先查找名为 `editor.c`、`screen.c` 和 `keyboard.c` 的文件并将它们编译为目标文件 (`editor.o`、`screen.o` 和 `keyboard.o`), 然后, 编译生成默认目标 `editor`。

实际的机制比这里所描述的要全面。目标文件 (`.o`) 可以从 C、Pascal 和 Fortran 等源代码中生成, 所以 `make` 也应去查找符合实际情况的相关文件。所以, 如果在工作目录下有 `editor.p`、`screen.p` 和 `keyboard.p` 三个 Pascal 文件 (`.p` 通常被认为是 Pascal 源代码的扩展名), `make` 就会激活 Pascal 编译器来编译它们, 而不用 C 编译器。因此, 如果出于某种原因而在项目中需要使用多种语言时, 就不能依靠隐式规则, 因为此时使用该规则所得到的结果可能会与期望的有所不同。

4.3.4 模式规则

通过定义用户自己的隐式规则，模式规则提供了扩展 make 的隐式规则的一种方法。模式规则类似于普通规则，但是它的目标必定含有符号“%”，这个符号可以与任何非空字符串匹配；为与目标中的“%”匹配，这个规则的相关文件部分也必须使用“%”。例如，下面的规则：

```
%o : %.c
```

告诉 make 所有形为 somename.o 的目标（object）文件都应从源文件 somename.c 编译而来。

与隐式规则一样，make 预定义了一些模式规则：

```
%o : %.c
      $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

与前面的例子相同，make 定义了一条规则，即任何 x.o 的文件都从 x.c 编译而来。每次使用该规则时，该规则用自动变量“\$<”和“\$@”来代替第一个依赖体和目标体。此外，变量\$(CC)，\$(CFLAGS)和\$(CPPFLAGS)的默认值如表 4.2 所示。

4.3.5 注释

在 makefile 中插入注释时，必须在注释前加上符号“#”。make 读到“#”后，它忽略该符号以及这一行余下的字母。注释可以出现在 makefile 的所有地方。但是，因为多数 shell 把“#”看作是元符号（通常也是注释符），所以在命令中加入注释时要特别小心。此外，实际上就 make 本身而言，一个只含注释的行就是一个空行。

4.4 命令行选项和参数

同多数 GNU 程序一样，make 也有丰富的命令行选项。表 4.3 列出了最常用的部分。

表 4.3 常用的 make 命令行选项

选项	说明
-f file	指定 makefile 的文件名
-n	打印将需要执行的命令，但实际上并不执行这些命令
-Idirname	指定被包含的 makefile 所在的目录
-s	在执行时不打印命令名
-w	如果 make 在执行时改变目录，打印当前目录名
-Wfile	如果文件已修改，则使用-n 来显示 make 将要执行的命令
-r	禁止使用所有 make 的内置规则
-d	打印调试信息

(续表)

选项	说明
-i	忽略 makefile 规则中的命令执行后返回的非零错误码, 此时, 即使某个命令返回非零的退出状态值, make 仍将继续执行
-k	如果某个目标编译失败, 继续编译其他目标。通常, make 在一个目标编译失败后终止
-jN	每次运行 N 条命令, 这里 N 是非零整数

4.5 调试 make

如果在使用 make 时遇到问题, -d 选项能够使 make 在执行命令时打印大量的额外调试信息。此时, 因为需要显示 make 内部所做的每一件事以及为什么做这些事的调试信息, 将会产生大量的输出。其中包括如下信息:

- 在重新编译时 make 需要检查的文件
- 被比较的文件以及比较的结果
- 需要被重新生成的文件
- make 想要使用的隐式规则
- make 实际使用的隐式规则以及所执行的命令

4.6 常见的 make 出错信息

这里列出使用 make 时可能遇到的最常用的出错信息, 完整文档请参见 make 使用手册或其信息页。

- No rule to make target 'target'. Stop makefile 中没有包含创建指定的 target 所需要的规则, 而且也没有合适的默认规则可用。
- 'target' is up to date 指定 target 的相关文件没有变化。
- Target 'target' not remade because of errors 在编译 target 时出错, 这一消息仅在使用 make 的 -k 选项时才会出现。
- command: Command not found make 找不到命令。通常是因为命令被拼写错误或者不在路径 \$PATH 下。
- Illegal option - option 在调用 make 时包含了不能被 make 识别的选项。

4.7 有用的 makefile 目标

除了前面提及的 clean, 编写 makefile 时还有一些常用的目标。名为 install 的目标把最

终的二进制文件，所支持的库文件或 shell 脚本，以及相关的文档移动到文件系统中与之相应的最终位置，并适当设置文件的权限和属主。此外，`install` 通常也编译程序，以及运行简单的测试以确认程序已正确编译。`uninstall` 目标则删除由 `install` 目标所安装的那些文件。如果需要，在设置 `install` 目标前存储系统当前的设置。

`dist` 目标可以用来生成准备发布的软件包。最低限度，`dist` 目标将删除编译工作目录中旧的二进制文件和目标文件并创建一个归档文件（如普通的压缩包），以便上载到万维网页或 FTP 站点。

为了方便其他开发者，可以用一个 `tags` 目标来创建或更新程序的标记表。如果程序的验证过程比较复杂，也可以创建一个单独的 `test` 或 `check` 目标来执行这一过程并显示适当的诊断信息。与之类似，`installtest` 或 `installcheck` 目标，通常被用来验证安装过程。当然，在此之前，`install` 目标必须已经成功地编译和安装了所需的程序。

4.8 小 结

本章介绍了 `make` 命令、作用和怎样编写简单实用的 `makefile`，并且还讨论了 `make` 规则和 `make` 的一些有用的命令行选项。通过学习本章，读者应当已能够使用 `make` 来管理软件项目的创建和维护过程。

第 5 章 创建可移植的自配置软件

Linux 的复杂起源以及存在多种 Linux 发布版本的现实情况需要一个灵活而且适应能力强的配置和编译环境。本章将介绍 GNU autoconf，这个工具可以对软件进行设置使之能够在包括许多非 Linux 系统在内的众多类型的系统配置中编译。

5.1 考虑可移植性

在开始学习 autoconf 之前，对创建 autoconf 这个工具的动机作些了解会对读者有所帮助。从最基本的层次来看，可移植性是指以这样的方式编写程序，即相同的源代码不必改动就能在多种平台上编译。本节将对可移植性的定义作详细说明，并且讨论在编写可移植性代码时需考虑的一些问题。

5.1.1 什么是程序的可移植性

开发能够运行在多种不同平台上的软件是一项需要很多技巧和努力的工作。仅仅创建能够在多种不同的 UNIX 类和 UNIX 系统上运行的程序也要做大量的工作。首先，代码自身必须是可移植的，可移植代码很少对运行时的硬件以及可以使用的软件库等有所假设。此外，如果编写的是 C 代码，为了保证最大的可移植性，则在语法上必须严格遵循 ISO/ANSI C 标准，或者把含有非标准 C 用法的程序隔离到尽可能少的模块中。

第二，开发者必须对不同系统的编译和运行环境，甚至是硬件体系结构有足够的了解。Linux 中普遍使用的 GNU（自由软件）软件，虽然也存在于许多其他的操作系统和硬件平台，但是不能够保证在那些操作系统上必然能够找到所需要的软件。此外，也可能存在以下情况：

- C 编译器可能不遵循 ISO 标准
- 函数库可能缺少关键的特性
- 系统服务的功能可能不同
- 文件系统所做的约定不同

在硬件方面，必须处理好高址结尾（big-endian）、低址结尾（little-endian）以及混合式的数据表示机制。除了 Intel 的 x86 系列处理器之外，还会遇到 PA-RISC、几种不同的 Sparc、驱动 Macintosh 计算机和 Apple 计算机的 Motorola 芯片（好几代不同的产品）、MIPS、Amiga 以及即将出现的 Intel Itanium 或 IA64 芯片。最终，开发人员不得不编写一个通用的 makefile 并且告诉用户如何编辑这个 makefile 文件以适应本地环境。

很清楚，要满足所有这些需求是一项让人感到畏惧的工作，但是 autoconf 能够减轻这项工作的负担。

5.1.2 移植性的线索和技巧

虽然 `autoconf` 能够让编写真正可移植代码的工作变得容易些，但是你仍然有任务要完成。一般说来，可移植的软件会对它编译和运行时的环境尽可能作出一些假设。例如，不会主动假定 `stdin` 和 `stdout` 是控制台，系统有 32 位彩色或者程序只能运行在 Caldera OpenLinux 系统上。

也就是说，我个人的观点认为你可以做出至少两种假设。第一，如果你正在编写 C 代码，那么可以（或者应该）假定使用 ANSI C 编译器。第二，如果你正在为 UNIX 或 Linux 环境开发软件，那么可以假定是为 POSIX 兼容环境编写程序。

事实证明，即便是上述假定也不是万无一失的。首先，人们可能会更聪明些，懂得 ANSI C 和 POSIX 兼容以外的知识。更严肃地看，仍然存在这样的环境，特别是嵌入式系统，没有针对这种环境的 ANSI 编译器。在这种情况下，比较公正的假定似乎是绝大多数 Linux 程序员都不会去编写嵌入式系统的软件。当然，本书并不只是针对这一类读者，但即便是这类读者也会在本书中找到有用的信息。

类似地，POSIX 标准以 UNIX 及其变体为基础。很明显，非 UNIX 系统，比如微软的 Windows 系列操作系统没有模仿类似 UNIX 的环境，所以基于 POSIX 的术语以及 UNIX 和 Linux 的术语和假定都没有很好地引入这类环境。

关键在于，如果你正在阅读本书，那么假定你的程序要在带有 ANSI 编译器并且兼容 POSIX 的环境下编译和运行是个合理的想法。

注意：虽然 Window NT 和它的后继产品都声称与 POSIX 兼容，那么试着在 NT 上编译一个使用了 POSIX 调用的程序。当你编辑好源代码并查看了编译器选项后，你就会得出这样的结论，NT 家族的 POSIX 兼容性往好说是微软的吹牛皮，往坏说是微软的神话。

怎样才能使程序具有可移植性呢？

- 尽可能避免针对特定系统的假定和方法。例如，不认定程序只在 OpenLinux 系统上运行，或者只在使用 RPM 包管理系统的系统上使用。
- 隔离依赖于系统的部分。如果使用了一种特殊的 GUI 环境，例如 `ncurses`，那么要把 GUI 部分的代码单独放到它自己的模块中。这样做可以使移植任务变得简单，比如让程序使用基于 X 的 GUI 环境或者另一种基于文本的 GUI S-Lang 而不是 `ncurses`。
- 尽可能复用已有的接口。为什么一定要重新创造已经存在的東西呢？从常用的数据库管理库，比如 Berkeley DB 或 GNU DBM 中选一种代替你自己的数据库管理库。
- 使用标准接口，比如多种 POSIX 标准；标准语言，比如 C 和 C++；以及标准库，比如标准 C 库、NAG 数学库和 `terminfo` 等常用库。

很遗憾，很难在这么少的篇幅给出详细步骤或提供软件移植的解决方案。实际上，移植性是一个复杂高深的主题，足够写一本书。如果对这个问题的研究感兴趣，在 <http://www.cs.wvu.edu/~jdm/research/portability/portbib.html> 上提供的参考书目是个很好的资源。

5.2 理解 autoconf

使用 autoconf 可以解决上一节所列的大部分问题。它生成一个能自动配置源代码包的 shell 脚本, 以使程序能够在许多不同品牌的 UNIX 和类 UNIX 系统上编译和运行。这些脚本通常被命名为 configure, 它们检查在当前系统中是否提供程序所需要的某些功能, 并在此基础上生成 makefile。而且, 如果把非标准的宏放入单个文件, autoconf 能够根据 configure 脚本执行后返回的测试结果来定义 (或不定义) 它们。那么通过在代码中测试这些宏, 用户的代码就有可能适应了主机系统上某种特性或功能存在的或者不存在的情况。最好的一点是, autoconf 生成的脚本是自包含的, 因此在目标系统上编译软件时不需要在其系统上安装 autoconf, 使用者所需要做的只是在软件发布版本的源程序目录中键入 ./configure。

为了生成 configure 脚本, 需要在源文件树的根目录下创建名为 configure.in 的文件。configure.in 调用一系列 autoconf 宏来测试程序需要的或用到的特性是否存在, 以及这些特性的功能。autoconf 包括很多预定义的宏, 用以测试常见的必要的特性。如果 autoconf 的内置宏不能够满足要求, 程序员可以使用第二组宏来建立自己需要的测试集; 同时, 如果需要, configure.in 也可以包括测试不常见的或特殊的功能所需要的 shell 脚本。要完成这些工作, 除了 autoconf (本章讨论的是 2.13 版) 外, 还需要至少 1.1 版的 GNU m4 程序, m4 通过展开输入文件中的宏来生成输出文件 (出于执行速度的考虑, autoconf 的作者 David MacKenzie 建议使用 1.3 版或更新的 m4 程序)。这两个软件的最新版本都可以从 GNU 的网站 <http://www.gnu.org> 或 FTP 站点 <ftp.gnu.org> 或者许多其他站点得到。而且, 多数的 Linux 发行版本中也都包含了这两个软件。

5.2.1 创建 configure.in

每个 configure.in 文件必须在开始所有测试前调用 AC_INIT, 并且在结束所有测试后调用 AC_OUTPUT。而事实上, 也只有这两个宏是必须的。AC_INIT 的语法如下:

```
AC_INIT(unique_file_in_source_dir)
```

unique_file_in_source_dir 是在源代码目录下的一个文件, 对 AC_INIT 的调用在所产生的配置脚本文件中生成一条 shell 命令, 通过检查 unique_file_in_source_dir 是否存在来验证当前目录是否正确。

AC_OUTPUT 创建名为 makefile 或其他名字 (可选) 的输出文件, 其语法如下:

```
AC_OUTPUT([file...[,extra_cmds[,init_cmds]])
```

其中 file 是用空格分隔的输出文件列表, 通过复制 file.in 到 file 来生成这些文件。extra_cmds 是一个命令列表, 附加在 config.status 之后, 在重新生成配置脚本时会用到它, init_cmds 也将插入到 config.status 中, 但其位置正好在 extra_cmds 之前。

5.2.2 构造文件

除了少数情况, 对 autoconf 宏的调用次序不会对结果产生影响 (我们将注意那些特殊情况), 也就是说, 下面的调用次序只是建议性质的, 而非必须:

AC_INIT

测试程序

测试函数库

测试头文件

测试类型定义

测试结构

测试编译器行为

测试库函数

测试系统调用

AC_OUTPUT

上述建议的调用顺序反映了一个事实，例如函数库的存在与否直接影响到是否包含相应的头文件，所以对头文件的检查要放在检查完函数库之后。类似的，一些系统服务取决于是否存在某些特殊库函数，而这些库函数只有当在头文件中声明了原型函数后才会被调用，并且，如果所需的函数库不存在，程序就不能在头文件中调用这些原型函数。除非确切地知道所做改动的意义，并且有足够的理由做这样的改动，否则最好不要改变对宏调用的建议次序。

在这里有必要注意一下 `configure.in` 的写法。每一个宏调用应该占据单独的一行，这是因为多数 `autoconf` 宏都需要一个新行来结束命令。在使用宏读取或设置环境变量时，可以把这些变量当作一个宏而放在同一行。

一个多参数的单宏调用可以超过这个每宏一行的规则。这时应该使用 `\` 来续行并且用 `m4` 所能识别的括号 `[]` 来括起所有参数。下面的两个宏调用是等价的：

```
AC_CHECK_HEADERS([unistd.h termios.h termio.h sgTTY.h alloca.h \
sys/itimer.h])
AC_CHECK_HEADERS(unistd.h termios.h termio.h sgTTY.h alloca.h sys
/timer.h)
```

第一个例子用一对 `[]` 把参数括起，并且使用 `\` (`\` 可以被 `shell` 而不是 `m4` 或 `autoconf` 所解释) 来表示续行；而第二个例子只是简单地把整个宏写在同一行内。

最后，可以使用 `m4` 的注释符号 `dnl` 在 `configure.in` 中插入注释。例如：

```
dnl
dnl This is an utterly gratuitous comment
dnl
AC_INIT(some_darn_file)
```

5.2.3 有用的 autoconf 工具

除了下一节将详细叙述的 `autoconf` 内置宏，`autoconf` 软件包也包含几个有用的脚本，可以帮助开发人员创建和维护 `configure.in`。在开始测试之前，可以用 Perl 脚本 `autoscan` 从源文件中抽取与函数调用和头文件有关的信息，并将其输出到 `configure.scan` 文件中。`autoscan` 的完整语法为：

```
autoscan [ --macrodir=dir ] [ --help ] [ --verbose ] [--version]
[ srcdir]
```

它没有必须要求的参数。srcdir 指定了扫描的目录。在大多数情况下，只在源代码树的根目录下执行 autoscan 就足够了。但是，在将该文件重命名或复制到 configure.in 之前，需要手工检查一下以确认是否遗漏了需要抽取的特性。ifnames 工具的功能与 autoscan 类似，它查找源文件中的预处理指令 #if、#elif、#ifdef 和 #ifndef。可以用它来增加 autoscan 的输出。ifname 的语法是

```
Usage: ifnames [ -h ] [ --help ] [ -m dir ] [ --macrodir=dir ]
[ --version ] [ file ... ]
```

关键的参数是 file ...，它是要扫描的源代码文件名的列表，列表可以包含一个或多个文件名。

本章的源代码目录中包含的源代码文件和多个支持文件都用于一个叫作 bogusapp.c（见代码清单 5.1）的程序，它是专门为了展示 autoconf 的特性而虚构的。在代码清单之后，简要介绍一下支持文件。

代码清单 5.1 bogusapp.c

```
#include <stdio.h>
#include <stdlib.h>
#ifdef HAVE_RESOLV_H
#include <resolv.h>
#endif /* HAVE_RESOLV_H */
#include "config.h"

int main(void)
{
    int retval;

#ifdef HAVE_MMAP
    fprintf(stdout, "have mmap()\n");
#else
    fprintf(stderr, "no mmap()\n");
#endif

if(HAVE_UTIME_NULL)
    fprintf(stdout, "utime() allows NULL\n");
else
    fprintf(stderr, "utime() doesn't allow NULL\n");

if(SYS_SIGLIST_DECLARED)
    fprintf(stdout, "sys_siglist() declared\n");
else
    fprintf(stderr, "sys_siglist() not declared\n");

#ifdef HAVE_NCURSES_H
    fprintf(stdout, "ncurses.h found\n");
#else
```



```

        fprintf(stderr, "ncurses.h not found\n");
    #endif

    if(HAVE_FCNTL_H)
        fprintf(stdout, "fcntl.h found\n");
    else
        fprintf(stderr, "fcntl.h not found\n");

    if(HAVE_SYS_FCNTL_H)
        fprintf(stdout, "sys/fcntl.h found\n");
    else
        fprintf(stderr, "sys/fcntl.h not found\n");

    #ifdef NLIST_NAME_UNION
        fprintf(stdout, "nlist.n_un member found\n");
    #else
        fprintf(stdout, "nlist.n_un member not found\n");
    #endif

    if(HAVE_VOID_POINTER)
        fprintf(stdout, "Yep, we have a usable void pointer
                        type\n");
    else
        fprintf(stderr, "Nope, no usable void pointer type\n");

    exit(EXIT_SUCCESS);
}

```

- **Makefile.in**——用于创建真正的 makefile 文件的模板
- **acconfig.h**——与特定系统相关的宏的集合，它随 autoconf 软件一起提供
- **bogusapp.c**——bogusapp 的源代码，这是个示例程序
- **config.h**——包含 bogusapp.c 中用到的所有宏的头文件
- **configure.in**——创建最终的 configure 脚本的模板
- **install.sh**——安装脚本，用在不带兼容 BSD 的 install 程序的系统上

在目录下运行 autoscan 产生的 configure.scan 如下：

```

$autoscan
$cat configure.scan
dnl Process this file with autoconf to produce a configure script.
AC_INIT(acconfig.h)

dnl Checks for programs.

dnl Checks for libraries.

dnl Checks for header files.
AC_HEADER_STDC

dnl Checks for typedefs, structures, and compiler characteristics.
dnl Checks for library functions.

```

AC_OUTPUT(Makefile)

这是 configure.in 文件的框架。但是，不要把它名字改成 configure.in，因为本章的源代码中已经包括了一个完整的 configure.in 文件！

ifnames 的输出如下：

```
$ ifnames *.c
HAVE_MMAP bogusapp.c
HAVE_NCURSES_H bogusapp.c
HAVE_RESOLV_H bogusapp.c
NLIST_NAME_UNION bogusapp.c
```

注意：输出包含了找到的条件名以及找出条件的文件名。把它和 autoscan 的输出以及完整的 configure.in 相比较，以保证没有遗漏任何条件或其他的预编译符号。

5.3 内 置 宏

在多数情况下，仅用 autoconf 内置宏就可以满足要求。每个内置的测试集在随后将细分为几个分别测试特定功能或进行更一般测试的宏。本节将简单介绍大部分内置测试，如果需要 autoconf 预定义测试的完整列表和描述，可以参考 autoconf 信息页。

5.3.1 候选程序测试

表 5.1 列出了一组用于检查指定程序的存在及其行为的测试，以便在需要从候选程序中选择时使用。虽然编译过程的配置相当复杂，但这些宏可以确认所需要的程序是否存在，以及如果存在它们是否能被正确调用，所以能给开发人员带来一定的灵活性。

表 5.1 候选程序测试集

测试	说明
AC_PROG_AWK	顺序检查 mawk、gawk、nawk 和 awk 是否存在，将输出变量 AWK 设置为所找到的第一个程序名
AC_PROG_CC	决定使用哪个 C 编译器，并设置输出变量 CC
AC_PROG_CC_C_O	决定编译器是否接受 -c 或 -o 选项，如果不接受，定义 NO_MINUS_C_MINUS_O
AC_PROG_CPP	把输出变量 CPP 设置为执行 C 预处理的命令
AC_PROG_INSTALL	把输出变量 INSTALL 设置为 BSD 兼容的 install 程序，或者是 install -sh
AC_PROG_LEX	查找 flex 或 lex，并把输出变量 LEX 设为结果
AC_PROG_LN_S	如果系统支持符号链接，则把变量 LN_S 设为 ln -s，否则设置为 ln
AC_PROG_RANLIB	如果 ranlib 存在，则设置输出变量 RANLIB 为 ranlib，否则设置为 “:”
AC_PROG_YACC	顺序检查 bison、byacc 和 yacc，并根据它找到的结果把输出变量 YACC 设为 bison -y、byacc 或 yacc

通常，表 5.1 中的宏用于建立所测试程序的路径或其所遵循的调用方法。以 AC_PROG_CC 为例，如果在某些目标系统上并没有 gcc，不要硬编码 gcc。此外，因为一些老的编译器（至少包括非 GNU 编译器）并不一定和 gcc 一样支持 -c 和 -o 选项，因此需要使用 AC_PROG_CC_C_O 测试宏。类似的，AC_PROG_LN_S 的存在是因为许多文件系统的实现不支持创建符号链接。

5.3.2 库函数测试

表 5.2 列出了用于测试特殊函数库的宏，这些宏先确认所测试的函数库是否存在，在存在库的情况下，再测试库中函数的参数表与相应函数是否有差别。一般而言，即使有最好的规划，函数库的变化也会逐渐积累到一定程度，使得新近版本的库不能和老版本兼容，有时还非常严重。表 5.2 中的测试宏使开发人员可以调整编译过程以适应这种不幸情况，在极端情形下，如果不能通过测试，也不能通过其他方式补救，那就只能放弃编译，等待系统升级了。

表 5.2 库函数测试集

测试	说明
AC_CHECK_LIB(lib, function[,action_if_found[,action_if_not_found[,other_libs]])	通过把一个 C 程序链接到函数库 lib 来判断在 lib 库中是否存在指定的函数。在测试成功时执行 shell 命令 action_if_found 或者在 action_if_found 为空时，在输出变量 LIB 中添加 -llib。action_if_not_found 把 lother_libs 选项传给 link 命令
AC_FUNC_GETLOADAVG	如果系统支持 getloadavg 函数，把获得该函数所必须的函数库添加到 LIBS 变量
AC_FUNC_GETPGRP	测试 getprgrp 是否需要参数，如果不需要，定义 GETPGRP_VOID，否则，getprgrp 需要一个进程 ID 作为其参数
AC_FUNC_MEMCMP	如果 memcmp 函数不存在，把 memcmp.o 添加到 LIBOBJS 中
AC_FUNC_MMAP	如果存在 mmap 函数，设置 HAVE_MMAP
AC_FUNC_SETPGRP	测试 setprgrp 是否需要参数，如果不需要，定义 SETPGRP_VOID，否则，setprgrp 需要两个进程 ID 作为其参数
AC_FUNC_UTIME_NULL	如果 utime(file,NULL)函数能把文件的时间戳设置为当前时间，定义 HAVE_UTIME_NULL
AC_FUNC_VFORK	如果 vfork.h 文件不存在，定义 vfork 为 fork
AC_FUNC_VPRINTF	如果存在 vprintf 函数，定义 HAVE_VPRINTF

AC_CHECK_LIB 是这组测试宏中最有用的一个，它给了开发人员一个机会来告诉使用者：“除非系统有所需要的函数库，否则该程序就不能正常运行。”其他测试宏的存在主要是为了弥补 BSD 和 AT&T UNIX 平台的差异。因为这两个 UNIX 分支在所支持的函数或函数的参数上有很大的区别，而 Linux 又继承了 BSD 和 AT&T UNIX，所以这些宏可以帮助开发人员来正确地配置软件。

5.3.3 头文件测试

头文件测试用于检查 C 头文件是否存在及其存在的位置。和表 5.2 中的那些宏一样，这些宏使开发人员能够考虑到不同系统上 UNIX 和 C 实现的差异。信不信由你，许多古怪的 UNIX 和老的 UNIX 系统以及类 UNIX 系统上都没有与 ANSI 兼容的 C 编译器，而其他一些系统也可能缺少 POSIX 兼容的系统调用。表 5.3 列出了这些测试。

表 5.3 头文件测试

测试	说明
AC_DECL_SYS_SIGLIST	如果 signal.h 或 unistd.h 定义了 sys_siglist，定义 SYS_SIGLIST_DECLARED
AC_HEADER_DIRENT	顺序检查头文件 dirent.h、sysdir/ndir.h、sys/dir.h 和 ndir.h 中是否定义了 DIR，并根据结果定义 HAVE_DIRENT_H、HAVE_SYS_NDIR_H、HAVE_SYS_DIR_H 或 HAVE_NDIR_H
AC_HEADER_STDC	如果系统支持 ANSI/ISO C 头文件，定义 STDC_HEADERS
AC_HEADER_SYS_WAIT	如果系统有 POSIX 兼容的头文件 sys/wait.h，定义输出变量 HAVE_SYS_WAIT

AC_HEADER_DIRENT 试图解决在 UNIX 以及类 UNIX 系统上存在大量不同文件系统所造成的问题，因为多数程序依赖于文件系统所能提供的服务，所以知道头文件的位置、能提供的功能以及调用约定就显得非常有用。AC_HEADER_STDC 检查是否存在兼容 ANSI/ISO 的头文件，但其结果并不能说明是否有这类编译器存在。

5.3.4 结构测试

结构测试在头文件中查找指定结构的定义、结构中某些成员是否存在及其类型。再重申一下，由于 UNIX 分裂为若干流派，所以不同的实现所提供的数据结构也不同。表 5.4 中列出的测试宏给了开发人员一个根据不同系统调整代码的机会。

表 5.4 结构测试

测试	说明
AC_HEADER_TIME	如果 time.h 和 sys/time.h 在一个程序中都存在，定义输出变量 TIME_WITH_SYS_TIME
AC_STRUCT_ST_BLKSIZE	如果 stat 结构有成员 st_blksize，定义输出变量 HAVE_ST_BLKSIZE
AC_STRUCT_ST_BLOCKS	如果 stat 结构有成员 st_blocks，定义输出变量 HAVE_ST_BLOCKS
AC_STRUCT_TIMEZONE	指出如何取得时区值。如果 tm 结构有成员 tm_zone，定义输出变量 HAVE_TM_ZONE；如果找到 tzname 数组，定义输出变量 HAVE_TZNAME

5.3.5 类型定义测试

表 5.5 列出了在头文件 sys/types.h 和 stdlib.h 中查找类型定义的测试宏。在某些类型可

能存在于一些系统中，而不存在于另一些系统中时，这些宏使得开发人员可以根据系统是否存在指定的类型定义来调整代码。

表 5.5 类型定义测试

测试	说明
AC_TYPE_GETGROUPS	根据传递给 <code>getgroups()</code> 的数组的基类型来设置 <code>GETGROUPS_T</code> 为 <code>gid_t</code> 或 <code>int</code>
AC_TYPE_MODE_T	如果 <code>mode_t</code> 没有定义，定义 <code>mode_t</code> 为 <code>int</code>
AC_TYPE_PID_T	如果 <code>pid_t</code> 没有定义，定义 <code>pid_t</code> 为 <code>int</code>
AC_TYPE_SIGNAL	如果 <code>signal.h</code> 中没有将 <code>signal</code> 定义为 <code>(void *)</code> ，定义 <code>RETSIGTYPE</code> 为 <code>int</code>
AC_TYPE_SIZE_T	如果 <code>size_t</code> 没有定义，定义 <code>size_t</code> 为 <code>unsigned</code>
AC_TYPE_UID_T	如果 <code>uid_t</code> 没有定义，定义 <code>uid_t</code> 和 <code>gid_t</code> 为 <code>int</code>

5.3.6 编译器行为测试

表 5.6 列出了测试编译器行为和特定主机结构特性的测试宏。这些测试宏给出大量与编译器和 CPU 有关的信息，程序员可以利用这些宏来调整代码以反映这些差别。

表 5.6 编译器行为测试

测试	说明
AC_C_BIGENDIAN	如果按高字节在前存储字，定义 <code>WORDS_BIGENDIAN</code>
AC_C_CONST	如果编译器不完全支持 <code>const</code> 声明，定义 <code>const</code> 为空
AC_C_INLINE	如果编译器不支持关键字 <code>inline</code> 、 <code>__inline__</code> 或 <code>__inline</code> ，定义 <code>inline</code> 为空
AC_C_CHAR_UNSIGNED	如果 <code>char</code> 是无符号数，定义 <code>CHAR_UNSIGNED</code>
AC_C_LONG_DOUBLE	如果主机编译器支持长双精度类型，定义 <code>HAVE_LONG_DOUBLE</code>
AC_C_CHECK_SIZEOF	把输出变量 <code>SIZEOF_Uctype</code> 定义为 C 或 C++ 预定义 <code>type</code> 类型的大小值 (<code>type [, cross-size]</code>)

5.3.7 系统服务测试

表 5.7 列出了检查操作系统服务及其功能的测试宏。不同的操作系统所提供的系统服务及其功能有很大不同，所以程序应该尽可能地处理好这种差异。

表 5.7 系统服务测试

测试	说明
AC_SYS_INTERPRETER	根据脚本是否以 <code>#!/bin/sh</code> 为提示符来设置 <code>shell</code> 变量 <code>ac_cv_sys_interpreter</code> 为 <code>yes</code> 或 <code>no</code>
AC_PATH_X	找到 X Window 头文件和库文件所在的路径，并设置 <code>shell</code> 变量 <code>x_includes</code> 和 <code>x_libraries</code> 为适当路径值，如果没有找到路径，则设置 <code>no_x</code>
AC_SYS_LONG_FILE_NAMES	如果系统支持长于 14 个字符的文件名，定义 <code>HAVE_LONG_FILE_NAMES</code>
AC_SYS_RESTARTABLE_SYSCALLS	如果系统调用会重启信号中断，定义 <code>HAVE_RESTARTABLE_SYSCALLS</code>

也许你会觉得奇怪，但是确实存在着把文件名限制在 14 个字符的文件系统，这甚至包括某些 UNIX 文件系统，AC_SYS_LONG_FILE_NAMES 使开发人员可以检测到这种野蛮的文件系统。此外，AC_PATH_X 的存在归因于那些不支持 X Window 系统的操作系统。

5.3.8 UNIX 变体测试

这一组测试宏用于测试特定的 UNIX 和类 UNIX 操作系统的特性。但正如 autoconf 的作者所说的：“这些宏存在只是鸡肋而已，它们将被更系统化的方法所取代，那些方法将测试系统所支持的函数以及环境，而不仅仅是系统类型 (34)”。表 5.8 列出了这些测试宏。

表 5.8 UNIX 变体测试宏

测试	说明
AC_AIX	如果宿主系统是 AIX，定义 _ALL_SOURCE
AC_DYNIX_SEQ	已废弃——用 AC_FUNC_GETMNTENT 代替
AC_IRIX_SUN	已废弃——用 AC_FUNC_GETMNTENT 代替
AC_ISC_POSIX	定义 _POSIX_SOURCE 以允许使用 POSIX 特性
AC_MINIX	在 MINIX 系统上定义 _MINIX 和 _POSIX_SOURCE 以允许使用 POSIX 特性
AC_SCO_INTL	已废弃——被 AC_FUNC_STRFTIME 代替
AC_XENIX_DIR	已废弃——被 AC_HEADER_DIRENT 代替

“我为什么要关心其中那些已废弃的宏呢？”，你也许会这么想。有两个理由。首先，你可能运行包含这些宏的 configure.in 文件，如果遇到这种情况，就可以把这些宏替换为相对应的宏。其次，这些宏被废弃仅仅是因为有了更具普遍意义的宏；但是，这些宏的存在也正好表明了有人量的现存代码仍然依赖于与操作系统的实现有关的特性，以及在不同实现的 UNIX 间仍然存在着差异。

提示： 要获得最新的有关被废弃宏信息的最简单方法是经常访问 GNU 的 FTP 站点或其他站点来跟踪 autoconf 发布版本中的 ChangeLog 文件的变化。

5.4 普通宏

autoconf 手册把下列宏作为新的测试的主要部分。多数情形下，这些宏测试编译器的行为，所以需要有一个可以被预处理、编译和链接（甚至执行）的测试程序，根据编译器对这个程序的输出和错误信息就可以确定这些测试成功与否。

```
AC_TRY_CPP([includes [,action_if_true[,action_if_false]])
```

这个宏把 includes 文件名传给预处理程序，如果预处理程序处理成功则执行 shell 命令 action_if_true，反之执行 action_if_false。

```
AC_EGREP_HEADER(pattern,header,action_if_found \
[,action_if_not_found])
```

这个宏可以用来在头文件 `header` 中查找 `egrep` 的表达式 `pattern`, 如果找到 `pattern`, 则执行 shell 命令 `action_if_found`, 反之执行 `action_if_not_found`。

```
AC_EGREP_CPP(pattern,program,[action_if_found \  
[,action_if_not_found]])
```

用预处理程序对 C 源代码 `program` 进行处理以查找 `egrep` 的表达式 `pattern`, 如果找到 `pattern`, 则执行 shell 命令 `action_if_found`, 反之执行 `action_if_not_found`。

```
AC_TRY_COMPILE(includes,function_body,[action_if_found \  
[,action_if_not_found]])
```

这个宏查找 C 或 C++ 编译器的某个语法特性。编译器将编译包含 `includes` 中的头文件并使用 `function_body` 中定义的函数的测试程序, 如果编译成功, 则执行 shell 命令 `action_if_found`, 反之执行 `action_if_not_found`。这个宏不执行链接, 可以用 `AC_TRY_LINK` 来测试链接情况。

```
AC_TRY_LINK(includes,function_body,[action_if_found \  
[,action_if_not_found]])
```

这个宏在 `AC_TRY_COMPILE` 之后增加链接测试。编译器将编译并链接其中包含 `includes` 中的头文件并使用 `function_body` 中定义的函数的测试程序, 如果链接成功, 则执行 shell 命令 `action_if_found`, 反之执行 `action_if_not_found`。

```
AC_TRY_RUN(program,[action_if_true[,action_if_false \  
[,action_if_cross_compiling]])
```

这个宏测试宿主系统的运行时行为。编译、链接和执行 C 程序 `program`, 如果 `program` 返回 0, 则执行 shell 命令 `action_if_true`, 否则执行 `action_if_false`。如果程序要编译为在另一类型的系统上运行, 则用 `action_if_cross_compiling` 替代 `action_if_found`。

```
AC_CHECK_PROG
```

测试在当前路径下是否存在指定程序 `program`。

```
AC_CHECK_FUNC
```

测试指定函数是否在 C 的链接函数库中存在。

```
AC_CHECK_HEADER
```

测试指定头文件是否存在。

```
AC_CHECK_TYPE
```

如果指定的类型没有被定义, 设置一个默认值。

5.5 一个带注释的 autoconf 脚本

在本节中将创建一个 `configure.in` 的示例文件。这个例子并不能配置一个实际能用的软件,只是用来演示许多在前一节讨论的宏,例子中的有些宏以前没讨论过,有些宏是 autoconf 的其他特性。

每一段代码后面讨论了这段代码的作用。

```
dnl Autoconfigure script for bogusapp
dnl Kurt Wall <kwall@kurtwerks.com>
dnl
dnl Process this file with 'autoconf' to produce a 'configure' script
```

第一个代码段是标准的 `autoconfig.in` 文件头,指出了这个 `configure.in` 脚本属于什么软件包、联系信息(通常是软件包的维护者)以及重新生成配置脚本的说明。

```
AC_INIT(bogusapp.c)
AC_CONFIG_HEADER(config.h)
```

接下来的两行调用了前面介绍过的 `AC_INIT` 函数,并且在源文件树的根目录下创建了一个名为 `config.h` 的头文件,其中只包含从实际的头文件中提取的预处理器符号。主要在源代码中包含这个头文件并使用其中的相关符号,实际的程序就能在每个可能的系统上平滑无缝地编译。autoconf 根据名为 `config.h.in` 的输入文件来生成 `config.h`,在 `config.h.in` 中包含了程序需要的所有 `#define` 指令。

怎样创建 `config.h.in`? 幸运的是,autoconf 自带了一个名为 `autoheader` 的 shell 脚本,这个脚本使用起来很方便。该脚本能生成 `config.h.in`。autoheader 通过读入 `configure.in`、作为 autoconf 软件一部分的 `acconfig.h` 文件和位于源代码树根路径下用于保存预处理符号的 `acconfig.h` 文件,生成 `config.h.in` 文件。在你开始抱怨又要创建另一个文件之前,告诉你一个好消息, `./acconfig.h` 只需包含在别处没有定义的预处理符号。更好地是,这些符号值都能为空。这个文件中只需要包含可以被 autoconf 和 autoheader 读取和使用的合法定义的 C 风格预处理符号。要创建 `config.h.in`,在创建了你的 `config.in` 文件之后在源代码目录下执行 `autoheader`。下面的代码段是用于 `bogusapp` 的 `acconfig.h` 文件。

```
/* Define this 1 if your compiler allows a (void *) function
return */
#define HAVE_VOID_POINTER 0

/* Define this 1 if your C compiler has a short_short_t type */
#define short_short_t 0

/* Define this 1 if your signal handling library support
sys_siglist */
#define HAVE_SYS_SIGLIST 0
```

正如在注释中看到的那样,要让这些宏起作用只要把它们值设置为 1 即可。此处把它们定义为 0 纯粹是为了方便性和一致性。

```
test -z "$LDFLAGS" && LDFLAGS="-I/usr/include" AC_SUBST(LDFLAGS)

dnl Tests for UNIX variants
dnl
AC_CANONICAL_HOST
```

AC_CANONICAL_HOST 报告从 GNU 观点看到的宿主机类型。它输出 `cpu-company-system` 形式的系统名称。例如，笔者的一个系统上，AC_CANONICAL_HOST 报告其类型为 `i686-unknown-linux`。

```
dnl Tests for programs
dnl
AC_PROG_CC
AC_PROG_LEX
AC_PROG_AWK
AC_PROG_YACC
AC_CHECK_PROG(SHELL, bash, /bin/bash, /bin/sh)
```

这一代码段按顺序判断并设置了编译器、词法分析器 `lexer`、`awk`、`yacc` 以及本地 `shell`。

```
dnl Tests for libraries
dnl
AC_CHECK_LIB(socket, socket)
AC_CHECK_LIB(resolv, res_init, [echo "res_init() not in libresolv"],
[echo "res_init() found in libresolv"])
```

“Test for libraries”的这一段代码展示了怎样为 `autoconf` 的宏编写自定义的命令。第二个 `AC_CHECK_LIB` 宏的第三个参数和第四个参数是两个 `shell` 命令，分别对应于前面讨论的 `action_if_found` 和 `action_if_not_found`。因为 `m4` 在引用和限定上的特殊性，建议用 `m4` 的引用字符(“`[`”和“`]`”)包含“`or`”的命令括起来，以免 `shell` 展开这类命令。

```
dnl Tests for header files
dnl
AC_CHECK_HEADER(killer.h)
AC_CHECK_HEADERS([resolv.h temio.h curses.h sys/time.h fcntl.h \
sys/fcntl.h memory.h])
AC_DECL_SYS_SIGLIST
AC_HEADER_STDC
```

以“`\`”结尾的一行说明了多参数续行的正确方式。前面已经介绍过，使用字符“`\`”告诉 `m4` 和 `shell` 需要续行，并且用 `m4` 的引用符号把整个参数括起来。

```
dnl Tests for typedefs
dnl
AC_TYPE_GETGROUPS
AC_TYPE_SIZE_T
AC_TYPE_PID_T

dnl Tests for structures
AC_HEADER_TIME
```

```

AC_STRUCT_TIMEZONE

dnl Tests of compiler behavior
dnl
AC_C_BIGENDIAN
AC_C_INLINE
AC_CHECK_SIZEOF(int, 32)

```

`AC_C_BIGENDIAN` 宏将产生一个警告，因为调用 `AC_TRY_RUN` 时没有设置默认值以允许交叉编译，可以忽略这个警告。

```

dnl Tests for library functions
dnl
AC_FUNC_GETLOADAVG
AC_FUNC_MMAP
AC_FUNC_UTIME_NULL
AC_FUNC_VFORK

dnl Tests of system services
dnl
AC_SYS_INTERPRETER
AC_PATH_X
AC_SYS_RESTARTABLE_SYSCALLS

```

`AC_SYS_RESTARTABLE_SYSCALLS` 宏将产生一个警告，因为调用 `AC_TRY_RUN` 时没有设置默认值以允许交叉编译，可以忽略这个警告。

```

dnl Tests in this section exercise a few of 'autoconf' 's generic
macros
dnl
dnl First, let's see if we have a usable void pointer type
dnl
AC_MSG_CHECKING(for a usable void pointer type)

```

现在情况开始变得有趣起来。基本上，普通宏允许你通过编写自己的宏对 `autoconf` 进行扩展。例如，`AC_MSG_CHECKING` 在屏幕上打印字符串“checking”，随后是一个空格以及传入的参数（在本例中是“for a usable void pointer type”。这个宏使你能像 `autoconf` 一样向用户报告当前的工作，让他们知道 `configure` 正在做什么。使用这个宏时最好显示地锁定屏幕）。

```

AC_TRY_COMPILE( [ ],
    [ char *ptr;
      void *xmalloc();
      ptr = (char *) xmalloc(1);
    ],
    [AC_DEFINE(HAVE_VOID_POINTER) AC_MSG_RESULT(usable void
      pointer) ],

```

留意 `AC_TRY_COMPILE` 宏。`autoconf` 能够把 C 代码嵌入到一个 C 程序框架中，并把

这个程序写入已生成的 `configure` 脚本中，以便在运行 `configure` 时编译这个程序；然后，`configure` 捕获编译器的输出并查找错误（如果读者通过在 `configure` 脚本中查找 `xmalloc` 来跟踪这个过程）。`AC_DEFINE(HAVE_VOID_POINTER)` 产生了一个名为 `HAVE_VOID_POINTER` 的预处理器符号（必须将它放置在 `./acconfig.h` 中，因为它并不在其他地方存在）。如果编译成功，`configure` 把 “`#define HAVE_VOID_POINTER 1`” 写入到 `config.h` 中并且在屏幕上打印 “usable void pointer”；如果编译失败则在 `config.h` 中写入 “`/* #undef HAVE_VOID_POINTER */`”，并显示 “no usable void pointer”。在你自己的源代码文件中，只需按如下的方式测试这个预处理器符号：

```
#ifdef HAVE_VOID_POINTER
/* do something */
#else
/* do something else */
#endif

dnl Now, let's exercise the preprocessor
dnl
AC_TRY_CPP(math.h, echo 'found math.h', echo 'no math.h - deep doo
doo!')
```

如果 `configure` 找到了头文件 `math.h`，它会在屏幕上显示 “found math.h”；否则它通知用户出现了一个问题。

```
dnl Next, we test the linker
dnl
AC_TRY_LINK([#ifndef HAVE_UNISTD_H
#include <signal.h>
#endif],
[char *ret = *(sys_siglist + 1);],
[AC_DEFINE(HAVE_SYS_SIGLIST), AC_MSG_RESULT(got sys_siglist)],
[AC_MSG_RESULT(no sys_siglist)])
```

这一段代码测试链接器。同样，因为 `HAVE_SYS_SIGLIST` 不是一个标准预处理器符号，你必须在 `./acconfig.h` 中声明它。

```
dnl Finally, set a default value for a ridiculous type
dnl
AC_CHECK_TYPE(short_short_t, unsigned short)
```

最后的测试只检查一种（希望的）不存在的 C 数据类型。如果确实没有，则将 `short_short_t` 定义为 `unsigned short`。读者可以到 `config.h` 中查找与 `short_short_t` 相关的 `#define` 指令来确认这个测试的结果。

```
dnl Okay, we're done. Create the output files and get out of here
dnl
AC_OUTPUT(Makefile)
```

在完成所有的测试以后就可以创建 `makefile` 了。`AC_OUTPUT` 把 `autoconf` 的测试结果

转换为编译器能够识别的形式，因此当用户在命令行键入 `make` 时，编译器就可以根据宿主系统的特性来生成可执行程序。为此，`AC_OUTPUT` 需要一个源文件做参数（在本例中是 `Makefile.in`）。

读者也许还记得在说明 `autoconf` 宏的用法时经常见到的类似“定义输出变量 `FOO`”的语句，`autoconf` 能够根据这些输出变量的值来设置 `makefile` 和 `config.h` 中的值。例如，如果找到 `tzname` 数组，`AC_STRUCT_TIMEZONE` 就定义变量 `HAVE_TZNAME`，此时，在 `configure` 创建的 `config.h` 文件中，也就会有 `#define HAVE_TZNAME 1` 这样的定义。这样，在源程序中就可以用如下的条件语句来封装使用 `tzname` 数组的代码。

```
if (HAVE_TZNAME)
    /* do something */
else
    /* do something else*/
```

类似地，在 `Makefile.in` 中有一系列形如“`CFLAGS=@CFLAGS@`”的表达式，而且 `configure` 根据测试的结果给所有这类“`@output_variable@`”赋予正确的值。在本例中，“`@CFLAGS@`”中包含调试和优化选项，默认是“`-g -O2`”。

在这个模板创建以后，在 `configure.in` 所在的目录（应在源代码树的根目录）下键入 `autoconf`。你可能会在屏幕看到在 `configure.in` 的第 48 行和第 63 行有两个警告，内容如下：

```
configure.in:48: warning: AC_TRY_RUN called without default to allow
cross compiling
configure.in:63: warning: AC_TRY_RUN called without default to allow
cross compiling
```

最终的结果是在当前工作目录下生成一个名为 `configure` 的 shell 脚本。要测试它，可键入“`./configure`”。图 5.1 显示了 `configure` 执行时的输出。

```
creating cache ./config.cache
checking host system type... Invalid configuration 'i686-pc-linux-gnu': machine
'i686-pc-linux' not recognized

checking for gcc... gcc
checking whether the C compiler (gcc -I/usr/include) works... yes
checking whether the C compiler (gcc -I/usr/include) is a cross-compiler... no
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
checking for flex... flex
checking for ywrap in -lfl... yes
checking for mawk... mawk
checking for bison... bison -y
checking for bash... /bin/sh
checking for socket in -lsocket... no
checking for res_init in -lresolv... yes
res_init() not in libresolv
checking how to run the C preprocessor... gcc -E
checking for killer.h... no
checking for resolv.h... yes
checking for termio.h... yes
checking for curses.h... yes
checking for sys/time.h... yes
checking for fcntl.h... yes
checking for sys/fcntl.h... yes
checking for memory.h... yes
checking for sys_siglist declaration in signal.h or unistd.h... yes
checking for ANSI C header files... yes
```

图 5.1 `configure` 正在运行

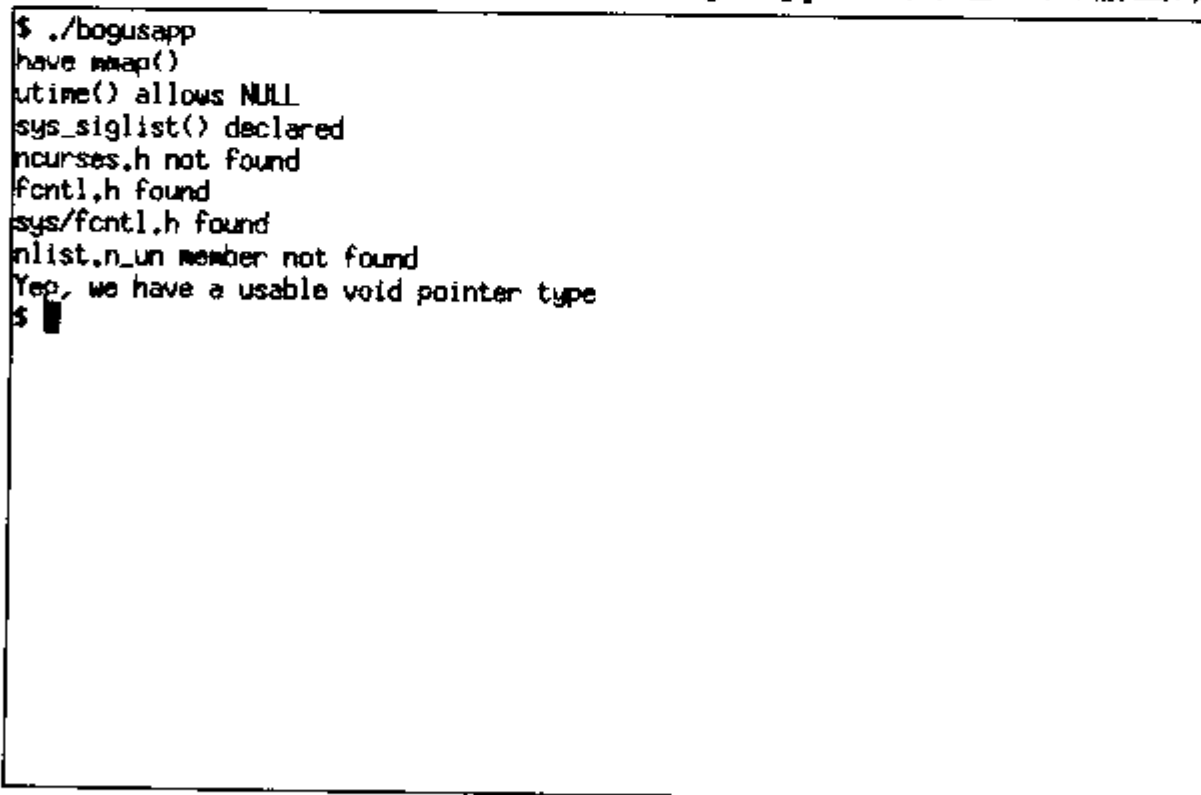
如果一切按照设计中的进行: `configure` 创建 `Makefile`、`config.h` 并且把所有动作记录在 `config.log` 中, 现在就可以在命令行键入 `make` 来测试所生成的 `makefile` 了。日志文件非常有用, 当 `configure` 不能产生正确的结果时, 就可以查看日志文件, 因为其中记录了 `configure` 在每一步试图去做的事。例如, 下面的日志文件代码段摘录了 `configure` 在查找 `socket` 函数时所采取的步骤 (参见 `configure.in` 的第 24 行)。

```
Configure:979: checking for socket in -lsocket
Configure:998: gcc -o conftest -g -O2 -I/usr/include conftest.c
-lsocket 1>&5
/usr/i386-linux/bin/ld: cannot find -lsocket
configure: failed program was:
#line 987 "configure"
#include "confdefs.h"
/* Override any gcc2 internal prototype to avoid an error. */
/* We use char because int might match the return type of a gcc2
   builtin and then its argument prototype would still apply. */
char socket();

int main() {
  socket()
  ; return 0; }
```

读者可以看到链接程序 `ld` 因找不到 `socket` 函数库 `libsocket` 而执行失败。此外, `log` 文件中显示的行号就是 `configure` 脚本正在执行的命令的行号。

最后, 在成功地构建了程序之后, 键入 “`./bogusapp`” 执行它。其输出结果见图 5.2。



```
$ ./bogusapp
have mmap()
utime() allows NULL
sys_siglist() declared
ncurses.h not found
fcntl.h found
sys/fcntl.h found
nlist.n_un member not found
Yep, we have a usable void pointer type
$
```

图 5.2 执行 `bogusapp`

虽然看起来有些冗长和无聊, 但是使用 `autoconf` 确实可以给软件开发人员带来很大好处, 尤其是程序要在不同的操作系统和硬件平台上移植或者允许用户根据自己系统的特性来定制软件的情况下。而且, 开发人员只需设置 `autoconf` 一次, 此后创建和维护自配置软件就很容易了。

5.6 小 结

本章详细讨论了 `autoconf`。在概述了 `autoconf` 的使用之后，介绍了 `autoconf` 的内置宏，这些内置宏可用来根据目标平台配置软件的编译过程。同时，也使读者了解到虽然不同操作系统的本质相同，但其实现上的不同足以使开发人员编写在这些系统上都能运行的软件的理想成为一场噩梦。最后，这里用一个例子逐步介绍了使用 `autoconf` 的整个过程：创建模板文件，生成 `configure` 脚本并使用该脚本生成 `makefile`。

第 6 章 比较和合并源代码文件

程序员经常需要快速区别两个文件的不同之处，或者合并两个文件。GNU 项目的 `diff` 和 `patch` 就是实现这两种功能的程序。本章的第一部分介绍怎样创建差异文件，这种文件说明了两个文件的不同之处。第二部分介绍了如何使用 `patch` 把源代码补丁文件应用到你的源代码上，补丁文件记录了对代码所做的修改。

6.1 使用 `diff` 命令比较文件

`diff` 命令是一组用来比较文件的命令中的一个。其他相关的命令包括 `cmp`、`wdiff`、`diff3` 和 `sdiff`，但是它们在编程时很少用到。

`diff` 的命令行选项和参数

`diff` 命令比较两个不同的文件或不同目录下的两个同名文件。在使用 `diff` 时，可以用选项来定制输出格式。6.3 节中要提及的 `patch` 程序将读取 `diff` 的输出和所比较文件中的一个来重新生成另一个。`diff` 手册的作者写道：“如果你认为 `diff` 是通过从一个文件中减去另一个来生成这两个文件的差别文件，那就可以认为 `patch` 是使用这个差别文件和其中的一个源文件来生成另一个源文件”。

在这里我们出于实用的目的，将只从程序员的观点出发讨论 `diff` 的用途，而忽略它的其他选项和功能。虽然文件比较是一个乏味的主题，但这个主题下却有大量的相关文档。如果读者需要查看 `diff` 选项的完整列表或者是与文件比较相关的理论，可以参见 `diff` 信息页 (`info diff`)。

`diff` 命令的一般语法为：

```
diff [options] srcfile dstfile
```

`diff` 在运行时试图找到在 `srcfile` 和 `dstfile` 里都一样的很多连续行，在碰到 `srcfile` 和 `dstfile` 里不一样的行时运行被打断，这些有差别的行称为块 (`hunk`)。因此，两个完全一样的文件不会有块，而两个完全不一样的文件会产生一个包含两个文件所有行的块。`diff` 输出中，`diff` 的比较行为和格式是由 `options` 控制的。`diff` 在两个文件间进行一行一行的比较。`diff` 产生几种不同的输出格式。表 6.1 描述了 `diff` 主要的命令选项和参数。

表 6.1 `diff` 的命令行选项和参数

选项	描述
-a	将所有的文件看作文本，即使文件看起来像是二进制的也不例外，并且进行逐行比较
-b	忽略块中空白数目的改变

(续表)

选项	描述
-B	忽略插入或删除空行造成的改变
-c	产生“上下文”(context)格式的输出生
-C [num]	产生“上下文”(context)格式的输出生, 显示块前后 num 行的内容, 如果不指定 num 的值, 则显示块前后 3 行的内容
-H	修改 diff 处理大文件的方式
-i	忽略大小写, 同样对待大写和小写字母
-I regexp	忽略插入或删除与正则表达式 regexp 匹配的行
-l	将输出结果通过 pr 命令处理加上页码
-p	显示出现块的 C 函数
-q	只报告文件是否不同; 不输出差别
-r	比较目录时, 进行递归比较
-s	报告两个文件相同(默认的行为是不报告相同的文件)
-t	输出时把 tab 扩展为空白
-u	产生“统一”(unified)格式的输出生
-U [num]	产生“统一”(unified)格式的输出生, 显示块前后 num 行的内容, 如果不指定 num 的值, 则显示块前后 3 行的内容
-v	打印 diff 的版本号
-w	逐行比较时忽略空白
-W cols	如果产生并排格式的输出生(参见-y), 让输出的每一列有 cols 个字符宽
-x pattern	当比较目录时, 忽略匹配模式 pattern 的任何文件和子目录
-y	产生并排格式的输出生

举几个例子能够便于理解其中的一些选项。本章的许多例子都以 hello.c 和 howdy.c 两个程序为基础, 它们分别如程序清单 6.1 和 6.2 所示。

程序清单 6.1 hello.c

```
#include <stdio.h>

int main(void)
{
    char msg[ ] = "Hello Linux programmer!";
    puts(msg);
    printf("Here you are, using diff.\n");
    return 0;
}
```

程序清单 6.2 howdy.c

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(void)
{
    char msg[ ] = "Hello, linux programmer, from howdy.c"

    puts(msg);
    printf("howdy.c says, 'Here you are, using diff.' \n ");
    exit(EXIT_SUCCESS);
}
```

使用 diff 的基本语法产生的输出（在下面“理解正规输出格式”部分所介绍的为正规格式）是：

```
$ diff hello.c howdy.c
1a2
> #include <stdlib.h>
5c6
<   char msg[ ] = "Hello, Linux programmer!";
---
>   char msg[ ] = "Hello, Linux programmer, from howdy.c!";
8c9
<   printf("Here you are, using diff.\n");
---
>   printf("howdy.c says, 'Here you are, using diff.'\n");
10c11
<   return 0;
---
>   exit(EXIT_SUCCESS);
```

结果看上去有点让人迷惑，所以在详细地讨论 diff 的选项和参数之前，需要知道怎样解释这样的输出，输出的外观取决于使用的输出格式。diff 能够产生几种输出格式，包括正规（normal，也是 diff 默认的输出格式）、上下文（context）、统一（unified）以及并排（side-by-side）4 种。

理解正规输出格式

之所以称为正规（normal）格式输出是因为这种格式只显示有差别的行，不会混入任何相同的行。它成为默认的输出格式的原因是为了遵守 POSIX 标准。正规格式很少用于发布软件补丁，但以此为基础对理解任何一种 diff 的输出格式很有用处。一般来说，正规块（normal hunk）的格式如下：

```
change_command
<srcfile line
<srcfile line...
...
>dstfile line
>dstfile line...
```

change_command 的格式如下：首先是一个来自 srcfile 的行号或以逗号隔开的行号范

围，然后是一个命令符，接下来是一个来自 dstfile 的行号或以逗号隔开的行号范围。其中的命令符可以为：

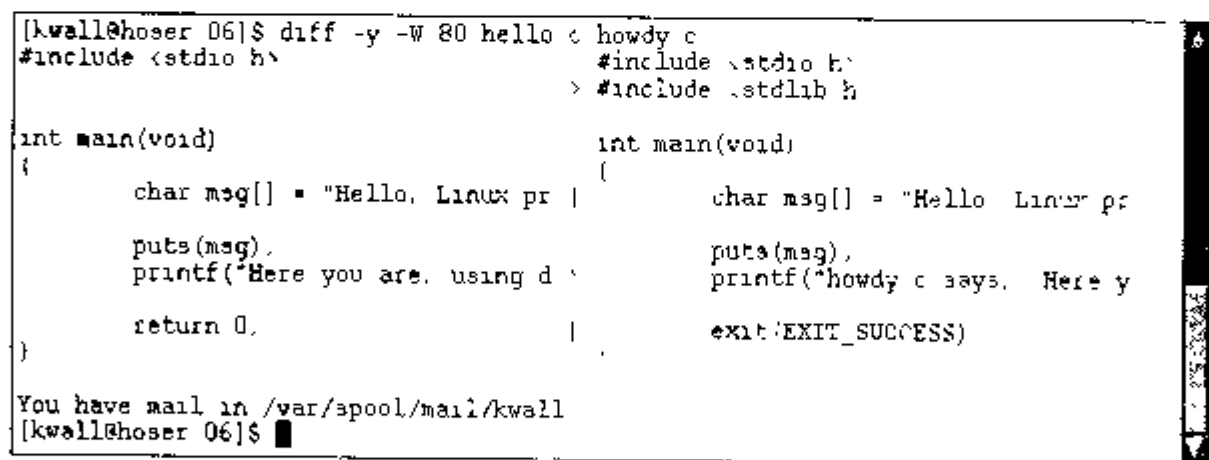
- a——添加
- d——删除
- c——更改

change_command 实际上是执行把 srcfile 转变为 dstfile 的 ed 命令。所以，对于上述的块来说就是把 hello.c 转换为 howdy.c，你必须做如下改动：

- 把 howdy.c 的第 2 行添加到 hello.c 的第 1 行后面
- 把 hello.c 的第 5 行变成 howdy.c 的第 6 行
- 把 hello.c 的第 8 行变成 howdy.c 的第 9 行
- 把 hello.c 的第 10 行变成 howdy.c 的第 11 行

理解并排输出格式

并排（side-by-side）格式虽然对于创建源代码补丁来说没有什么用处，但是用它直接比较源代码文件比较容易，因为它把 srcfile 和 dstfile 的内容并排显示在屏幕上。并排输出的行通常有 130 列，正好和老式的单行打印机匹配，但是不适合在终端屏幕上显示，因为终端屏幕通常为 80 列（字符）宽。于是，要取得完整效果，必须让终端屏幕更宽才行。图 6.1 显示了在正常宽度（80 列）的终端窗口中的并排输出格式。



```
[kwall@hoser 06]$ diff -y -W 80 hello.c howdy.c
#include <stdio.h>                                #include <stdio.h>
                                                    > #include <stdlib.h>

int main(void)                                    int main(void)
{
    char msg[] = "Hello, Linux pr |             char msg[] = "Hello, Linux pr
    puts(msg);                                     puts(msg);
    printf("Here you are, using d |             printf("howdy c says, Here y
    return 0;                                     |             exit(EXIT_SUCCESS)
}
You have mail in /var/spool/mail/kwall
[kwall@hoser 06]$
```

图 6.1 查看 diff 的并排输出格式

从图 6.1 可以看到，对两个文件在视觉上做快速对比，尽管截断了行，但还是清楚地显示出了两个文件哪里不同怎样不同。注意使用 -W 选项的 diff 命令可以指定输出列的宽度。diff 接受这个宽度并分给两个文件，给每个文件大约 40 列而且截断放不下的行。字符 “>” 表示该行在 dstfile 但不在 srcfile 里。类似地，字符 “<” 表示该行在 srcfile 而不在 dstfile 里。字符 “|” 标记出两个文件不相同的行。

理解上下文输出格式

以前曾提到过，在发布软件补丁时很少（可能从不）使用正规和并排的块格式。但 diff 产生的上下文（context）或统一（unified）的块格式是创建补丁所采用的格式。为了产生上下文的差异文件（它们称为 context diff 的原因是它们显示出了有差别的行的上下文内

容), 可使用 `diff` 的 `-c` 或 `-C [num]` 选项。如表 6.1 所述, `-c` 在显示每个有差别的行时同时还显示该行上下 3 行内容, 而 `-C [num]` 显示该行上下 `num` 行的内容, 如果没有指定 `num` 的数值, 则显示 3 行的内容。程序清单 6.3 用前面介绍的 `hello.c` 和 `howdy.c` 两个文件演示了 `diff` 的上下文格式。

提示: 和大多数 GNU 程序一样, `diff` 也支持长选项, 也就是以两个连字符 “--” 开头, 后面跟着更容易记忆的名字的选项。例如, 创建一个上下文 `diff` 文件的长选项是 “--context=[num]”。虽然 GNU 长选项更容易记住, 但却要输入更多的字符。可以查看手册或 `info` 页面了解 `diff` 或其他任何 GNU 命令的长选项。

程序清单 6.3 上下文输出举例

```
$ diff -c hello.c howdy.c
*** hello.c    Web Aug 9 21:02:42 2000
--- howdy.c    Web Aug 9 21:04:30 2000
*****
*** 1,12 ***
    #include <stdio.h>

    int main(void)
    {
!       char msg[ ] = "Hello, Linux programmer!";
        puts(msg);
!       printf("Here you are, using diff.\n");
!       return 0;
    }
--- 1,13 ---
    #include <stdio.h>
+ #include <stdlib.h>

    int main(void)
    {
!       char msg[] = "Hello, Linux programmer, from howdy.c!";
        puts(msg);
!       printf("howdy.c says, 'Here you are, using diff.'\n");
!       exit(EXIT_SUCCESS);
    }

$
```

上下文块的格式采用以下一般形式:

```
*** srcfile srcfile_timestamp
--- dstfile dstfile_timestamp
*****
*** srcfile_line_range****
    srcfile line
```

```

srcfile line...
--- dstfile_line_range
dstfile line
dstfile line..

```

开头两行标明了作比较的文件，第三行把标识信息同其余的输出内容分隔开来。输出的其余部分由一个或多个块所组成。每个块举出一个文件有差别的地方，并且带有 3 行（默认情况）上下文。上下文行以两个空格开头，而有差别的行以一个表示差异类型的字符开头，后跟一个空格。这个特殊字符可以是以下其中之一：

- + ——向 srcfile 添加一行以创建 dstfile。
- - ——从 srcfile 删除一行以创建 dstfile。
- ! ——在 srcfile 改变一行以创建 dstfile。srcfile 中标记“!”的每一行或一段，在 dstfile 中相应的每一行或一段也标记“!”。

每个块（hunk）都用一长串最多 15 个星号和下一块（hunk）分隔开来。

注意： patch 命令需要至少两行上下文才能正常工作。所以在为了发布软件补丁而产生上下文差异文件的时候，至少取得两行上下文。

接下来分析所创建的输出，diff 检测到了一个块，范围包括 hello.c 的从 1 到 12 行和 howdy.c 的从 1 到 13 行。每个文件有 3 行不同，用“!”标记出来。字符“+”标明向 hello.c 添加一行#include <stdlib.h>就能创建 howdy.c。

理解统一输出格式

统一格式是对上下文格式的修改版本，它不显示重复的上下文而且还用其他办法压缩输出内容。统一格式以下面的开头来标识要比较的文件：

```
--- srcfile srcfile_timestamp
```

其后是一个或多个块（hunk），格式如下：

```

@@ srcfile_range dstfile_range @@
line_from_either_file
line_from_either_file..

```

以@@开头的每一行都标志一个块的开始。在块中，上下文行以空格开头，而有差别的行以“+”或“-”开头，以表示相对于 srcfile 在此位置上添加或删除一行。你可以使用 -U [num] 选项改变上下文行数，num 是要显示的上下文行数。命令 diff -u hello.c howdy.c 产生的输出如下：

```

--- hello.c   Web Aug  9 21:02:42 2000
+++ howdy.c  Web Aug  9 21:04:30 2000
@@ -1,12 +1,13 @@
#include <stdio.h>
+#include <stdlib.h>

int main(void)

```

```

{
-   char msg [] = "Hello, Linux programmer!";
+   char msg [] = "Hello, Linux programmer, from howdy.c!";

    puts(msg);
-   printf("Here you are, using diff.\n");
+   printf("howdy.c says, 'Here you are, using diff.'\n");
-   return 0;
+   exit(EXIT_SUCCESS);
}

```

正如你所看到的那样，统一格式的输出更紧凑，因为没有重复的上下文行弄乱显示，所以易于理解。在本例中的上下文差异文件(context diff)里有一个包含了 hello.c 的从 1~12 行和 howdy.c 的从 1~13 行的块。对这一输出进行翻译，用语言来描述怎样把 hello.c 转变成 howdy.c:

- 紧挨着 #include <stdio.h> 一行之后加入 #include <stdlib.h>
- 紧挨着前半个大括号之后，删除

```
char msg[] = "Hello, Linux Programmer!";
```

并加入

```
char msg[] = "Hello, Linux Programmer, from howdy.c!";
```

- 紧挨着 puts(msg); 之后，删除

```
printf("Here you are, using diff.\n");
```

并加入

```
printf("howdy.c says, 'Here you are, using diff.'\n");
```

- 紧挨着最后一个空行之后，删除

```
return 0;
```

并加入

```
exit(EXIT_SUCCESS);
```

笔者认为，虽然统一格式既紧凑又容易阅读，但是统一格式的差异文件却有一个缺点：目前，只有 GNU diff 能产生统一格式的差异文件而且只有 GNU patch 能理解统一格式。所以，如果你要向没有或不能用 GNU diff 和 GNU patch 的系统发布软件补丁，则不要使用统一格式。而是使用标准的上下文格式。

在了解过 diff 的输出格式之后，下面准备看看它的一些命令行选项。虽然 diff 最适合比较纯文本，然而如果你希望，它也可以比较二进制文件并且以二进制格式显示差异文件。要判别两个二进制文件的差别，只需把它们的名字作为参数传递给 diff:

```

$ diff hello howdy
Binary files hello and howdy differ

```


如果要显示有差别的行，则使用 `-a` 选项。但是需注意这样做会输出毫无意义的内容，它们可能会破坏当前的终端会话。在这种情况下，比较明智的做法是将输出重定向到一个文件：

```
$ diff -a hello howdy > diffs
```

要查看两个文本文件是否不同但又不显示差异之处，可以使用 `diff` 的 `-q` 选项：

```
$ diff -q hello.c howdy.c
Files hello.c and howdy.c differ
```

假如你想忽略某种差别。实现这个目的的做法是使用 `-I regexp` 选项。如表 6.1 所述，这个表达式告诉 `diff` 忽略同正则表达式 `regexp` 相匹配的插入或删除行。下面的例子使用了 `-I regexp` 来忽略匹配“include”的行；目的是不显示添加或删除头文件的行：

```
$ diff -u -I include hello.c howdy.c
--- hello.c      Web Aug  9 21:02:42 2000
+++howdy.c      Web Aug  9 21:04:30 2000
@@ -2,11 +3,11 @@

    int main(void)
    {
-       char msg[ ] = "Hello. Linux programmer!";
+       char msg[ ] = "Hello. Linux programmer, from howdy.c!";

        puts(msg);
-       printf("Here you are, using diff.\n");
+       printf("howdy.c says, 'Here you are, using diff.\n'");
-       return 0;
+       exit(EXIT_SUCCESS);
    }
}
```

如果你把这里的输出同演示统一输出格式的例子做一比较，就会注意到涉及头文件的行都没有了。如果你只对两个文件的不同部分感兴趣，以这种方式消除不希望产生的输出是非常方便的。

另一组有用的 `diff` 命令行选项能改变它对空白的处理方式。`-b` 选项让 `diff` 忽略输入文件中空白数量的变化；`-B` 让 `diff` 忽略删除或插入空行的改动；`-w` 在逐行比较时忽略空白的变化。`-b` 和 `-w` 的区别在哪里？`-b` 忽略的是输入文件之间空白数量上的变化，而 `-w` 则忽略在原本没有空白的地方添加的空白。

考虑下面两行文字：

```
Here is a line with space between words.
```

```
Here is a line with space between words .
```

`diff -b` 能检测到“words”和“.”之间加入的空格，而 `diff -w` 则检测不到。

6.2 理解 diff3 命令

当两个人同时修改一个公用文件时, diff3 就会发挥作用。它比较两个人做出的两套修改内容, 创建第 3 个文件保存合并后的输出结果, 并且指出双方修改的冲突之处。diff3 的语法是:

```
diff3 [options] myfile oldfile yourfile
```

oldfile 是派生出 myfile 和 yourfile 的共同源文件。而这个[options]参数随后再介绍。程序清单 6.4、6.5 和 6.6 引入了 sigrot.1、sigrot.2 和 sigrot.3, 它们是一个 shell 脚本的 3 个版本, 这个 shell 能够轮换附加在 email 消息和 Usenet 文章上的签名。sigrot.3 和 sigrot.1 相比, 除了在脚本的末尾有一条 return 语句之外, 其他完全相同。如果你不理解脚本的 shell 语法, 也不必担心, 它们在这里只是为了演示 diff3 的用法。shell 编程将在第 30 章中进行介绍。

程序清单 6.4 sigrot.1

```
#!/bin/bash
# sigrot.sh
# Version 1.0
# Rotate signatures
# Suitable to be run via cron
#####

sigfile=signature

old=$(cat num)
let new=$(expr $old+1)

if [ -f $sigfile.$new ]; then
    cp $sigfile.$new .$sigfile
    echo $new > num
else
    cp $sigfile.1 .$sigfile
    echo 1 > num
fi
```

程序清单 6.5 sigrot.2

```
#!/bin/bash
# sigrot.sh
# Version 2.0
# Rotate signatures
# Suitable to be run via cron
#####

sigfile=signature
srcdir=$HOME/doc/signatures
srcfile=$srcdir/$sigfile
```

```

old=$(cat $srcdir/num)
let new=$(expr $old+1)

if [ -f $srcfile.$new ]; then
    cp $srcfile.$new $HOME/.$sigfile
    echo $new > $srcdir/num
else
    cp $srcfile.1 $HOME/.$sigfile
    echo 1 > $srcdir/num
fi

```

程序清单 6.6 sigrot.3

```

#!/bin/bash
# sigrot.sh
# Version 3.0
# Rotate signatures
# Suitable to be run via cron
#####

sigfile=signature

old=$(cat num)
let new=$(expr $old+1)

if [ -f $sigfile.$new ]; then
    cp $sigfile.$new . $sigfile
    echo $new > num
else
    cp $sigfile.1 . $sigfile
    echo 1 > num
fi

return 0

```

可以想像，由于需要处理 3 个输入文件，diff3 的输出将非常复杂。因此，diff3 仅显示这些文件的不同行。其中，若某些行在 3 个输入文件中都不相同，则包含这些行的 hunk 被称为 3 路 hunk，此外，使用 2 路 hunk 表示只在两个文件中有差别的行。3 路 hunk 用“===”标识，而 2 路 hunk 则在“===”后加上 1、2 或 3 来指出引起不同的那个文件。除此之外，diff3 在列举 hunk 的同时给出了生成这些 hunk 所需的一个或多个命令（仍旧使用 ed 形式）。这些命令如下：

- file:la 该 hunk 出现在第 l 行后，但在 file 中不存在这个 hunk，所以如果要依据 file 生成其他文件，必须加入在第 l 行后这个 hunk。
- file:rc 该 hunk 由 file 中的第 r 行组成，因此在生成其他文件时必须对该行进行指定的修改。

diff3 中，为了区分 hunk 与命令，hunk 以两个空格开始。例如，

```
$ diff3 sigrot.2 sigrot.1 sigrot.3
```

产生如下的输出（在这里因为空间的原因对输出做了截断）：

```
====
1:3c
  # Version 2.0
2:3c
  # Version 1.0
3:3c
  # Version 3.0
====1
1:9,10c
  srcdir=$HOME/doc/signatures
  srcfile=$srcdir/$sigfile
2:8a
3:8a
====1
1:12c
  old=$(cat $srcdir/num)
2:10c
3:10c
  old=$(cat num)
...
```

第一个 hunk 是 3 路 hunk，其他的都是 2 路 hunk。从 sigrot.1 或 sigrot.3 中生成 sigrot.2 时，必须把从 sigrot.2 中来的下面两行添加到 sigrot.1 或 sigrot.3 的第 8 行后：

```
srcdir=$HOME/doc/signatures
srcfile=$srcdir/$sigfile
```

类似地，若要依据 sigrot.2 来生成 sigrot.1，必须把 sigrot.1 的第 10 行改为 sigrot.2 中来的第 12 行。

如前所述，得到的结果相当复杂。为了避免出现这种情况，可以使用 -m 或 --merge 选项来告诉 diff3 对文件进行合并，然后再手工对结果排序：

```
$ diff3 -m sigrot.2 sigrot.1 sigrot.3 > sigrot.merged
```

该命令合并文件，标记冲突的上下文并输出结果保存到 sigrot.merged 文件中，程序清单 6.7 给出了该输出文件。可以看到，这个命令所产生的输出处理起来比较简单，因为只须注意不同文件的差别之处，这些差别分别用“<<<<<<”、“|||||”或“>>>>>>”标记。

程序清单 6.7 使用 diff3 合并选项产生的输出

```
#!/usr/local/bin/bash
# sigrot.sh
<<<<<<< sigrot.2
# Version 2.0
||||| sigrot.1
```

```

# Version 1.0
=====
# Version 3.0
>>>>>> sigrot.3
# Rotate signatures
# Suitable to be run via cron
#####

sigfile=signature
srcdir=$HOME/doc/signatures
srcfile=$srcdir/$sigfile

old=$(cat $srcdir/num)
let new=$(expr $old+1)

if [ -f $srcfile.$new ]; then
    cp $srcfile.$new $HOME/.$sigfile
    echo $new > $srcdir /num
else
    cp $srcfile.1 $HOME/.$sigfile
    echo 1 > $srcdir/num
fi

return 0

```

“<<<<<<” 标记对应 myfile, “>>>>>>” 对应 yourfile, “|||||” 对应 oldfile。在本例中, 只需要最新的版本号, 为了成功合并 3 个版本, 将删去标记行和 1.0 及 2.0 版本指定的行。

6.3 准备源代码补丁

在 Linux 中, 多数软件是以二进制 (可以直接运行) 或源代码格式发布的。在用源代码方式发布软件时, 可以是完整的代码包, 也可以是 diff 生成的补丁。GNU 项目 patch 工具能把 diff 文件合并到系统中现有的源代码树, 从而得到新的软件版本。下面将讲述 patch 的命令行选项, 使用 diff 生成补丁的方法以及在 patch 中使用补丁的方法。

与多数 GNU 项目工具一样, patch 也是一个稳定、强大而且多用途的工具。它可以读取的 diff 文件格式既包括标准格式或上下文格式, 也包括复杂的、GNG 定义的统一格式。此外, patch 可以根据需要从补丁文件中去掉首尾行, 因此可以直接用 email 或 Usenet 上的文章来安装补丁, 而不用执行预先编辑。

6.3.1 patch 的命令行选项

表 6.2 列出了常用的 patch 选项。使用 patch --help 或查看 patch 的 info 页面可以得到完整的选项列表。

表 6.2 patch 选项

选项	含义
-c	把输入的补丁文件看作是上下文格式的差异文件
-d dir	把 dir 设置为解释补丁文件名的当前目录
-e	把输入的补丁文件看作是 ed 脚本
-F num --fuzz=NUM	把非精确匹配的 fuzz 因子设置为 NUM 行
-l	把不同的空字符序列视为相同
-n	把输入的补丁文件看作是正规格式的差异文件
-pnum --strip=NUM	剥离文件名中的前 NUM 个目录成分
-R	假定在生成补丁的命令中交换了老文件和新文件的次序
-s	除非方式错误, 否则保持缄默
-t	执行过程中不要求任何输入
-u	把输入的补丁文件看作是统一格式的差异文件
-v	显示 patch 的版本信息并退出

在多数情况下, patch 程序可以确定补丁文件的格式, 当它不能识别时, 可以使用 -c、-e、-n 或者 -u 选项来指定输入的补丁文件的格式。前面已经提到过, 只有 GNU patch 可以创建和读取统一格式的差异文件, 因此, 除非能够确定补丁所面向的只是那些使用 GNU 工具的用户, 否则应该使用上下文的差异格式来生成补丁。同时, 为了使 patch 程序能够正常工作, 需要上下文的行数至少是 2。

fuzz 因子 (-F NUM 或者 --fuzz=NUM) 设置在定位正确的位置应用补丁时 patch 程序能忽略的最大行数。它的默认值是 2, 而且不能大于差异文件中上下文的行数。如果直接使用从 email 消息或 Usenet 文章中抽取的补丁文件, 那么邮件或新闻组的客户端程序可能会“友好地”把空格转换为 tab 或者把 tab 转换为空格。如果出现了这种情况, 那么在应用补丁文件时会遇到麻烦, 可以使用 patch 程序的 -l 选项来忽略空白。

有时候, 程序员在创建一个差异文件时会把文件名的顺序弄反。正确的顺序应该是 old-file new-file。如果 patch 程序碰到的补丁文件是用颠倒的顺序 (也就是 new-file old-file) 创建的, 它会认为这个补丁文件是一个逆向补丁 (reverse patch)。为了以正常的次序来使用逆向补丁, 可以指定 patch 程序使用 -R 选项。实际上, 如果 patch 程序检测到逆向补丁, 它会通知用户并且应用 -R 选项。用户也可以用 -R 选项把已经应用过的补丁再抵销掉。

patch 程序不但通用性好、功能强大, 而且还比较小心谨慎。当 patch 程序运行时, 它会对将要改动的每个源文件做备份, 在备份文件名的末尾加上 .orig 作后缀。如果 patch 程序不能应用某个块 (hunk), 它会用补丁文件中存储的文件名加上 .rej (拒绝) 后缀来保存该块。

6.3.2 创建补丁

使用 diff 创建补丁时, 要在命令行指定输出格式为上下文或统一 diff, 并且在 diff 命令行中按老文件先于新文件的顺序输入文件名, 输出文件名的后缀应当是 .diff 或 .patch。例如, 在根据 sigrot.1 和 sigrot.2 来生成补丁时, 可以用如下命令行生成一个上下文格式的 diff:

```
$ diff -c sigrot.1 sigrot.2 > sigrot.patch
```

或者使用

```
$ diff -u sigrot.1 sigrot.1 sigrot.2 > sigrot.patch
```

来创建统一格式的 diff。如果源代码树内包含子目录,则在使用 diff 时指定 -r(recursive) 选项以告诉 diff 在创建补丁文件时遍历所有子目录。

6.3.3 应用补丁

使用上面这个补丁的命令如下:

```
$ patch -p0 < sigrot.patch
```

-pnum 选项指定使用补丁前补丁中所包含的文件名中需要剥离的“/”的重数。例如,如果补丁中的文件名是/home/kwall/src/sigrot/sigrot.1,则-p1的结果是 home/kwall/src/sigrot/sigrot.1;-p4的结果是 sigrot/sigrot.1;-p 则剥去了除最终文件名之外的所有部分,得到 sigrot.1。

如果在安装完补丁后发现错误,只要简单地再原命令中加上-R 选项后再安装一次该补丁就能得到原来的文件:

```
$ patch -p0 -R < sigrot.patch
```

可以看到, diff 和 patch 的使用并不困难。虽然需要了解有多种文件格式以及命令的工作方式,但实际的使用却相当简单和直接。就如许多其他 Linux 命令一样,可以从中学到的东西很多,但有效地使用这些命令却不必知道所有的细节。

6.4 小 结

本章介绍了 diff、diff3 和 patch 命令。其中 diff 和 patch 是在创建和应用源代码补丁时最常用的工具。本章还介绍了 diff 不同的输出格式。标准的输出格式是上下文格式,因为大多数 patch 程序都能理解它。在从事实际开发工作时,读者会发现本章的内容是 Linux 软件开发工具包中的一个重要部分。

第 7 章 使用 RCS 和 CVS 控制版本

版本控制是指跟踪和管理源代码文件变化的自动过程。为什么需要版本控制？有很多理由：其一，也许有一天你对源代码作了关键改动，删除了老的文件并且忘记了所做改动的确切位置；其二，跟踪关于当前版本，下一版本以及修改过的错误的情况等信息是冗长乏味并且容易出错的事情；其三，也许你的同事不经意间修改了你的代码，会使得你不得不在备份磁带上疯狂查找以找回合适的版本。最终，也许你在某个早上一觉醒来，会对自己说：“版本控制，确实是一件非常必要的事情”。本章讨论两种版本控制系统：修订控制系统（Revision Control System, RCS）和并发版本系统（Concurrent Version System, CVS）。

7.1 基本术语

在开始介绍之前，我们先看一下表 7.1，它列出了本章将用到的术语。这些术语的使用极为频繁，因此希望读者在往下看之前确信自己已经理解了它们在 RCS 和版本控制中的确切含义。

表 7.1 版本控制术语

名称	说明
RCS File	在 RCS 目录下的文件，由 RCS 控制，并通过 RCS 命令存取。一个 RCS 文件包含某一特殊文件的所有版本。通常，RCS 文件的扩展名是.v
Working file	从 RCS 源代码库（即 RCS 目录）中检索到的一个或多个文件，放置在当前工作目录下，并能够被编辑
Lock	以编辑目的取回工作文件时别人就不能同时编辑这个文件。此时，文件由第一个编辑它的人锁定
Revision	源文件的一个特定版本，用数字标识。Revision 的编号从 1.1 开始，并依次递增，除非强制指定修订版本号

RCS 可以管理文件的多个版本，通常这些文件都是程序的源代码，但这不是必须的（笔者就用 RCS 来管理本书的不同修订版本）。RCS 自动处理各版本的存储、检索、更改日志、存取控制、发行管理、修订标识和合并；并且，由于它只跟踪文件的变化，所以只需要最小的磁盘空间。

注意：本章实例假定读者使用 5.7 版的 RCS。使用 `rcs -v` 命令可以确定 RCS 的版本号。

7.2 使用修订控制系统 (RCS)

简单是 RCS 的魅力之一。人们可以使用少量的命令来完成大量的工作。本节讨论 `ci`、`co` 和 `ident` 命令以及其他的 RCS 关键字。

7.2.1 RCS 基本用法

初次接触 RCS 的用户经常被其表面上的复杂性所吓退。放松！RCS 的入门出奇地简单。在读完本节的内容后，你会发现 80% 的 RCS 用法都包含了 `ci` 和 `co` 这两条命令，这两条命令分别用于把文件存入源代码库以及从源代码库取出源代码的检测。本节还要介绍 RCS 的关键字，它们可以让你向自己的源代码以及编译后的二进制文件中嵌入标识信息。

`ci` 和 `co`

只使用 `ci` 和 `co` 两条命令以及一个名为 RCS 的目录就可以完成 RCS 的许多工作。`ci` 代表“检入”(check in)，即在 RCS 目录下保存一个工作文件；而 `co` 代表“检出”(check out)，用于从 RCS 源代码库检索出 RCS 文件。首先，创建一个 RCS 目录：

```
$ mkdir RCS
```

如果在当前工作目录下存在 RCS 目录，RCS 命令就会使用这个目录。此外，RCS 目录也被称为源代码库 (repository)。接下来，在建好的 RCS 目录下创建一个名为 `yo.c` 的源代码文件，参见程序清单 7.1。

程序清单 7.1 `yo.c`——RCS 的基本用法

```
/* $Id$
 * yo.c - Code to demonstrate RCS usage
 */
#include <stdio.h>

int main(void)
{
    printf("yo, Linux programmer!");
    return 0;
}
```

执行命令 `ci yo.c`。RCS 会要求输入这个文件的描述信息，并将其复制到 RCS 目录下，然后删除原来的文件。“删除原来的文件？”是的。不用担心，你可以用命令 `co yo.c` 来从 RCS 中取回这个文件，此时取回的文件就是工作文件。需要注意的是，现在工作文件是只读的；如果要编辑它，则必须锁定它。使用 `co` 的 `-l` 选项 (`co -l yo.c`) 可以锁定指定的文件。`-l` 的含义是锁定，这在表 7.1 中已经解释过了。

```
$ ci yo.c
RCS/yo.c,v    <--    yo.c
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
```

```
>> Source code file to demo basic RCS usage.
>> .
initial revision: 1.1
done
$ co -l yo.c
co: RCS/yo.c,v: No such file or directory
[kwall@hoser 07]$ co -l yo.c
RCS/yo.c,v --> yo.c
revision 1.1 (locked)
done
$
```

注意： 当你首次检入一个文件时，还可以使用 `-i` 选项，告诉 RCS，特别是 `ci` 命令，这是初始检入该文件。

为了了解版本控制的工作方式，现在对工作文件做一些修改。如果刚才你没有做好准备，则应该先检出并锁定文件 (`co -l yo.c`)。现在可以对文件做任意修改，但建议在 `printf` 的字符串参数尾部添加 “`\n`”，因为 Linux（以及大部分 UNIX）在控制台输出后不会自动添加换行符，这一点与 DOS 或 Windows 不同。

```
printf("Yo, Linux Programmer!\n");
```

接着把修改后的文件存入 RCS，此时 RCS 会把版本号递增为 1.2，并要求输入与此次修改相关的描述信息，然后把所做的修改并入到 RCS 文件中，再删除工作文件。为了防止检入操作中删除（这一操作比较烦人）工作文件，可以在使用 `ci` 时指定 `-l` 或 `-u` 选项。

```
$ ci -l yo.c
RCS/yo.c,v <-- yo.c
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.' or end of file:
>> Added newline
>> .
done
$
```

`-l` 和 `-u` 选项在和 `ci` 连用时，都会在检入过程完成后对该文件执行一次隐式的检出操作。`-l` 选项锁定文件好让用户可以编辑它，而 `-u` 选项则检出工作文件的一个未上锁的只读版本以防被意外编辑。

除了 `-l` 和 `-u` 选项，`ci` 和 `co` 还接受另外两个非常有用的选项：`-r` 选项，能够让用户指定检出或检入的文件版本，而 `-f` 选项能强制 `ci` 和 `co` 执行检入和检出操作。使用 `-r` 告诉 RCS 你希望使用的文件版本。默认情况下，RCS 假定用户希望使用文件最后修订的版本；`-r` 选项则覆盖了这个默认值。例如，`ci -r2 yo.c`（这等价于 `ci -r2.1 yo.c`）创建了 `yo.c` 的 2.1 版本；`co -r1.7 yo.c` 检出 `yo.c` 的 1.7 版，而不管工作目录下的最高版本号。

`-f` 选项强制 RCS 覆盖当前的工作文件。默认情况下，如果在工作目录中存在同名的工作文件，则 RCS 的操作就会失败。因此，如果想要放弃当前的工作文件，就可以使用 `co -l`

-f yo.c, 这样就丢弃了所有未存回 RCS 的修改, 并从一个已知良好的源文件开始工作。使用 ci -f 将强制 RCS 保存文件, 即使这个文件并没有被修改过。

正如你所期望的那样, RCS 的命令行选项可以组合起来使用, 而且 RCS 对不兼容的选项做了适当处理。为了检出并锁定 yo.c 的特定版本, 可以使用 co -l -r2.1 yo.c。类似地, ci -u -r3 yo.c 把 yo.c 存回 RCS, 指定其修订版号为 3.1, 然后从 RCS 取回只读的 3.1 版工作文件保存到当前工作目录。

RCS 关键字

RCS 关键字是一些特殊的类似于宏的记号, 可以用来在源代码、目标文件或二进制文件中插入和维护标识信息。这些记号的形式是 \$KEYWORDS\$。当一个包含 RCS 关键字的文件被检出时, RCS 把 \$KEYWORDS\$ 扩展为 \$KEYWORD: VALUE\$。

\$Id\$

例如, 程序清单 7.1 顶部的特殊字符串 \$Id\$ 就是一个 RCS 关键字。在第一次检出 yo.c 时, RCS 把它扩展为如下字符串:

```
$Id: yo.c,v 1.2 2000/08/13 16:04:26 kwall Exp kwall $
```

在你的系统上, 其中大部分域会有不同的值; 而且, 如果在检出文件时加了锁, 在 Exp 后会列出你的登录名。

\$Id\$ 字符串的格式如下:

```
$Id: filename revision date time author state locker $
```

表 7.2 说明了每个域的含义。

表 7.2 关键字 \$Id\$ 的域

域	描述
filename	RCS 文件的名称, 不包含路径
revision	RCS 文件的修订号
date	本次修订检入源代码库的日期
time	本次修订检入源代码库的时间
author	检入本次修订的作者全名
state	文件状态
locker	如果 RCS 文件被锁定了, 锁定文件者的登录名

\$Log\$

通常, 在源代码中维护一个变动日志很有用。RCS 关键字 \$Log\$ 可以实现这个功能。RCS 把 \$Log\$ 关键字置换为检入过程中用户所提供的日志信息。但是, RCS 只是在先前的日志消息上面插入新的消息, 而不是用最新的消息取代以前的消息。程序清单 7.2 给出了一个例子来说明文件经过多次检入后, RCS 在检出文件时怎样扩展 \$Log\$ 关键字。

程序清单 7.2 多次检入后的\$Log\$关键字

```

/* $Id: yo.c, v 1.5 2000/08/13 16:52:30 kwall Exp kwall $
 * yo.c - Code to demonstrate RCS usage
 *
 *-----Revision History-----
 * $Log: yo.c,v $
 * Revision 1.5 2000/08/13 16:52:30 kwall
 * Added pretty box for the revision history
 * Revision 1.4 2000/08/13 16:51:09 kwall
 * Added args to main for command line processing
 *
 * Revision 1.3 2000/08/13 16:50:34 kwall
 * Added the Log keyword
 * -----
 */
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Yo, Linux programmer!\n");
    return 0;
}

```

使用\$Log\$关键字,我们就可以在编辑文件的同时方便地看到以往对这个文件做过的修改。按从上往下的顺序,最近所做的修改列在最前面。

其他 RCS 关键字

表 7.3 列出了其他的 RCS 关键字以及 RCS 对这些关键字的扩展方法。

表 7.3 RCS 关键字

关键字	描述
\$Author\$	检入该版本的用户的登录名
\$Date\$	该版本检入的日期和时间,使用 UTC 格式
\$Header\$	RCS 文件的全路径名、版本号、日期、时间、作者、状态和加锁者(在文件被加锁的情形下)
\$Locker\$	锁定该版本的使用者的登录名(如果没有被锁定,该域值为空)
\$Name\$	用于检出该版本的符号名(如果存在的话)
\$RCSfile\$	不包含路径的 RCS 文件名
\$Revision\$	该版本的版本号
\$Source\$	RCS 文件的全路径名
\$State\$	该版本的状态: Exp(实验版)、Stab(稳定版)或 Rel(发行版)。默认值是 Exp

ident 命令

ident 命令能够在所有类型的文件中定位并显示 RCS 关键字。开发人员利用这个功能就可以找出程序的特定发布版本中所用到的各个模块的版本号。为了说明这一点，我们可以创建如程序清单 7.3 所示的源代码文件。

程序清单 7.3 ident 命令

```
* prn_env.c - Display values of environment variables
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static char rcsid[] = "$Id$\n";

int main(void)
{
    extern char **environ;
    char **my_env = environ;

    while(*my_env) {
        printf("%s\n", *my_env);
        my_env++;
    }

    exit(EXIT_SUCCESS);
}
```

prn_env.c 这个程序遍历了包含在 environ 数组（头文件 unistd.h 里定义）中的一组环境变量，并且显示环境变量的所有值（参考 man(5) environ 了解更多信息）。语句 static char rcsid[]="\$Id\$\n"，利用 RCS 对关键字的扩展创建了一个静态文本缓冲区来保存 \$Id\$ 的实际值，以便 ident 能够从编译后的程序中抽取该值。

使用 -u 选项（`ci -u prn_env.c`）在 RCS 中检入 prn_env.c 文件，然后编译和链接这个程序（使用 `make prn_env.c` 或者 `gcc -Wall prn_env.c -o prn_env`）。期间忽略编译器给出的那个 rcsid 变量定义而未用的警告信息。如果乐意可以运行一下 prn_env 程序，然后执行命令 `ident prn_env`。如果一切顺利，得到的输出结果如下：

```
$ident prn_env
prn_env:
$Id: prn_env.c, v 1.1 2000/08/13 17:18:25 kwall Exp $
```

如前所述，\$Id\$ 关键字被展开为其实际值，而 GCC 则把这个值编译进了二进制文件。为了证实这一点，在源代码文件中找到 \$Id\$ 串，并将它与 ident 的输出作比较；可以看到，这两个字符串严格匹配。

ident 可以从源代码、目标文件或二进制文件中抽取形如 \$KEYWORD: VALUE \$ 的字符串。它甚至可以处理原始的二进制数据文件和内存转储（core dump）文件。实际上，因为 ident 查找所有能够和 \$KEYWORD: VALUE \$ 模式相匹配的字符串，所以其中也可以使

用非 RCS 关键字。这使得开发人员能够在程序中嵌入附加信息（例如公司名称）等。在模块中内嵌信息是一种有用的手段，因为这样就可以把问题隔离到特定的代码模块中了。这一特性的巧妙部分在于 RCS 能够自动更新标识串——对程序员和项目管理人员而言，这确实是个奖赏。

7.2.2 找出 RCS 文件间的不同

如果你需要知道一个工作文件和它所对应的 RCS 文件之间的差别，那么就可以使用 `rcsdiff` 命令。`rcsdiff` 使用 `diff(1)` 命令（已经在第 6 章中讨论过了）来比较文件。使用 `rcsdiff` 最简单的形式是 `rcsdiff filename`，`rcsdiff` 把源代码库中最新版本的 `filename` 和工作目录中的 `filename` 进行比较。你也可以用 `-r` 选项来指定特定的版本。

现在看看示例程序 `prn_env.c`。检出它的一个不加锁的版本，将其中的 `static char` 缓冲定义及其随后的空行删除。结果如下所示（当然，`Id` 行会有变化）：

```
/* $Id: prn_env.c,v 1.1 2000/08/13 17:18:25 kwall Exp kwall $
 * prn_env.c - Display values of environment variables
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    extern char **environ;
    char **my_env = environ;

    while(*my_env) {
        printf("%s\n", *my_env);
        my_env++;
    }

    exit(EXIT_SUCCESS);
}
```

现在执行命令 `rcsdiff prn_env.c`。RCS 编译和显示的输出如下：

```
$ rcsdiff prn_env.c
=====
RCS file: RCS/prn_env.c,v
retrieving revision 1.1
diff -r1.1 prn_env.c
8,9d7
< static char rcsid [] = "$Id: prn_env.c, v 1.1 2000/08/13 17:18:25
    kwall Exp kwall $\\n";
<
$
```

正如在第6章所学到的那样, diff 的输出意味着版本 1.1 中的第 8 和第 9 行如果没有删除应该出现在 prn_env.c 的第 7 行。

使用 -r 选项检查特定的版本, 将 prn_env.c 检入源代码库 (ci prn_env.c), 然后再从源代码库中检出并加锁 (co -l prn_env.c), 在结束 while 循环的括号下紧跟着加入一条语句 sleep(5); 最后再把这第三个版本用 -u 选项存回源代码库 (ci -u prn_env.c)。现在在源代码库中保留了三个不同版本的 prn_env.c。要证实这一点, 可以使用 rlog 命令 (rlog 命令将在 7.2.3 节中讨论):

```
$ rlog prn_env.c

RCS file: RCS/prn_env.c,v
Working file: prn_env.c
head: 1.3
branch:
locks:strict
access list:
symbolic names:
keyword substitution:kv
total revisions:3;   selected revisions: 3
description:
File to demonstrate rcsdiff
-----
revision 1.3
date: 2000/08/13 17:45:16; author:kwall; state: Exp; lines: +2
-1
Added a sleep statement
-----
revision 1.2
date: 2000/08/13 17:44:37; author:kwall; state:Exp; lines: +1 -3
Removed the rcsid buffer
-----
revision 1.1
date: 2000/08/13 17:43:41; author: kwall; state: Exp;
Initial revision
=====
$
```

使用 rcsdiff 比较文件的特定版本的一般格式为:

```
rcsdiff [-rrevision1 [ -frevision2 ]] filename
```

首先, 比较版本 1.1 和工作文件。

```
$ rcsdiff -r1.1 prn_env.c
=====
RCS file: RCS/prn_env.c,v
retrieving revision 1.1
diff -r1.1 prn_env.c
```

```

1c1
< /* $Id: prn_env.c, v 1.1 2000/08/13 17:43:41 kwall Exp $
---
> /* $Id: prn_env.c,v 1.3 2000/08/13 17:45:16 kwall Exp $
8,9d7
< static char rcsid = "$Id: prn_env.c, v 1.1 2000/08/13 17:43:41 kwall
Exp $\n";
<
18a17
>     sleep(5);
$

```

接着，比较版本 1.2 和版本 1.3。

```

$ rcsdiff -r1.2 -r1.3 prn_env.c
=====
RCS file: RCS/prn_env.c,v
retrieving revision 1.2
retrieving revision 1.3
diff -r1.2 -r1.3
1c1
< /* $Id: prn_env.c,v 1.2 2000/08/13 17:44:37 kwall Exp $
---
> /* $Id: prn_env.c,v 1.3 2000/08/13 17:45:16 kwall Exp $
16a17
>     sleep(5);
$

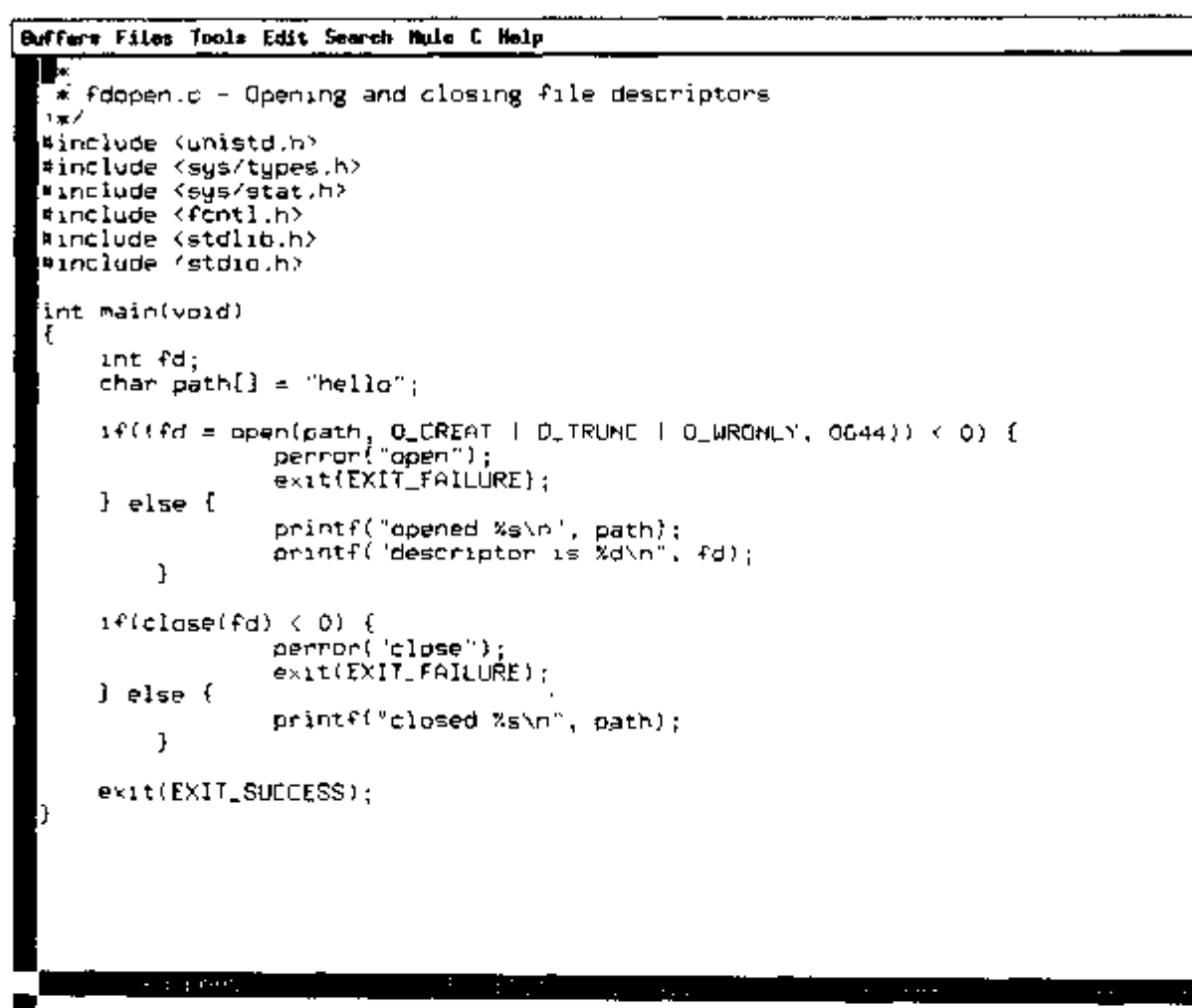
```

在需要比较工作文件与 RCS 文件的差别或者准备把多个版本合并成为一个版本的情况下，`rcsdiff` 是一个有用的工具。第 6 章讨论的 `diff` 的命令行选项和参数对应 `rcsdiff` 也同样适用。

对于 Emacs 迷而言，Emacs 有一个更好的版本控制工具 VC，它能够支持 RCS 和 CVS。例如，如果要把当前正在编辑的文件首次检入源代码库（称为用 RCS 注册一个文件），键入 `C-x v i`。在 Emacs 会话里工作时，要把当前文件检入或检出 RCS 源代码库，只需键入 `C-x v v` 或 `C-x C-q` 并根据提示输入信息。所有的 Emacs 版本控制命令的前缀都是 `C-x v`。图 7.1 显示了 GNU Emacs 使用 RCS 时进行注册的画面。

提示：记号 `C-x` 的意思是同时按下 `Control` 键和小写的 `x` 键。

Emacs 中的 VC 模式在很大程度上增强了 RCS 的基本功能。如果读者对 Emacs 感兴趣，可以深入了解一下 Emacs 的 VC 模式。



```
*
* fdopen.c - Opening and closing file descriptors
*/
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int fd;
    char path[] = "hello";

    if((fd = open(path, O_CREAT | O_TRUNC | O_WRONLY, 0644)) < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    } else {
        printf("opened %s\n", path);
        printf("descriptor is %d\n", fd);
    }

    if(close(fd) < 0) {
        perror("close");
        exit(EXIT_FAILURE);
    } else {
        printf("closed %s\n", path);
    }

    exit(EXIT_SUCCESS);
}
```

图 7.1 使用 GNU Emacs 中的 RCS 注册文件

7.2.3 其他 RCS 命令

除了 `ci`、`co`、`ident` 和 `rcsdiff`，RCS 组件还包括 `rlog`、`rcsclean` 和 `rcsmerge`，当然还有 `rcs`。这些命令可以增强开发人员对源代码的控制，合并或删除 RCS 文件，查看日志信息以及执行其他的管理功能。

`rcsclean`

`rcsclean` 所做的工作与其名称一致：清除 RCS 工作文件。其基本语法是：

```
rcsclean [options] [file ...]
```

`options` 指定你要使用的 RCS 选项，`file` 指明你要删除的文件。一个无选项的 `rcsclean` 命令将删除那些在取出后没有更改的工作文件。使用 `-u` 选项可以先解锁所有已加锁的文件，然后再删除没有更改的那些工作文件。使用 `-rM.N` 格式可以删除指定的版本。`M` 指主版本号，`N` 指次版本号。例如：

```
$ rcsclean -r2.3 foobar.c
```

从当前工作目录中删除 `foobar.c` 的 2.3 版。

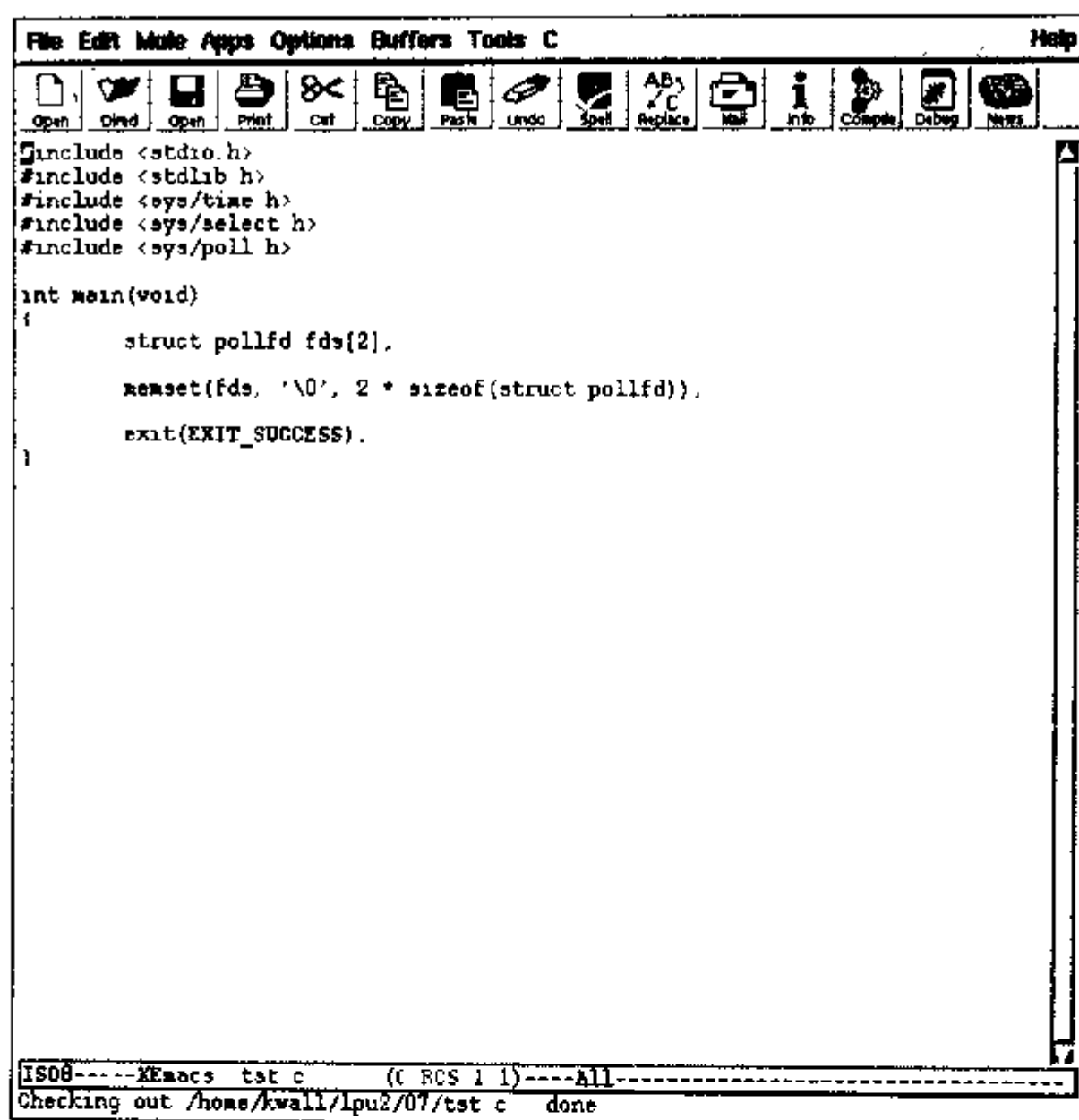


图 7.2 使用 XEmacs 中的 RCS 注册文件

rlog

rlog 可以打印文件存储在 RCS 源代码库中的日志消息和其他信息。例如，`rlog pm_env.c` 将显示所有版本的 `pm_env.c` 的日志信息。使用 `-R` 选项，则 **rlog** 只显示文件名；使用 `rlog -R RCS/*` 命令可以列出源代码库中的所有文件（当然，也可以直接使用 `ls -l RCS` 来做到这一点）；使用 `-L` 选项，即 `rlog -R -L RCS/*` 可以列出所有被加锁的文件；而使用 `-l` 选项可以列出所有被某个指定用户加锁的文件，例如，如果该指定用户名为 `gomer`，则命令如下：

```
$rlog -lgomer RCS/*
```

rcs

rcs 主要是一个管理性命令。但是它通常在两种情况下很有用。如果你检出了一个未上锁（只读）的文件并且做了修改，而你又不想放弃修改，命令 `rcs -l filename` 就能够检出上锁的 `filename` 文件，同时又不覆盖工作文件。

你还可以使用 **rcs** 命令检出一个由别人上锁的文件。如果你需要消除一个文件上面的锁，而这个文件是由别人检出的，可以使用 `rcs -u filename` 命令。文件会被解锁，同时原来加锁的人将收到你发出的一条消息，解释你为什么消除锁定的原因。

rcs 另一个有用的特性是它能够改变日志信息。还记得每次检入一个文件时，都可以输入一条检入信息来说明修改内容或工作内容吧。如果检入信息有录入或其他错误，或者仅

仅想增加一条附加信息，那么可以使用-m 选项。

```
$rcs -mrev:msg
```

rev 是你想要修正或改动的消息的版本号，而 msg 是要增加的正确的或附加的信息。

rcsmerge

rcsmerge 可以合并多个版本来生成一个单独的工作文件。其语法如下：

```
$ rcsmerge -rancestor -rdescendant working_file -p >merged_file
```

descendant 和 working_file 都必须来源于 ancestor。-p 选项告诉 rcsmerge 输出到 stdout（标准输出，一般是屏幕）而不是覆盖 working_file 文件。通过把输出重定向到 merged_file，我们可以检查合并的结果；因为虽然 rcsmerge 每次合并文件都尽力而为，但其结果仍是不可预料的。-p 选项可以使你能够检查这种不可预料性。

与 RCS 有关的更多信息，可以参见下列帮助页：rcs(1)、ci(1)、rcsintro(1)、rcsdiff(1)、rcsclean(1)、rcsmerge(1)、rlog(1)、rcsfile(1)和 ident(1)。

7.3 使用并发版本系统（CVS）

CVS 是最流行的开放源代码版本控制系统。它既可以用于一台单独的计算机、一个局域网，也可以利用其客户机/服务器的体系结构在因特网上使用。CVS 是建立在 RCS 的基础上的，所以前一节学到的 RCS 的知识在此也同样适用。但是，正如你将要学到的那样，CVS 对 RCS 的功能作了很大的扩展。

7.3.1 同 RCS 相比的优点

CVS 成为版本控制问题更好的解决方案的原因有几个。第一，它比 RCS 更适合于管理多目录的项目，因为它使用了单一的主代码树，而不是像 RCS 那样依赖多个目录。结果 CVS 构建项目的理念更直观。CVS 的第二个优点是它能处理分布式项目（distributed projects）。在这样的项目中，分别处于地理上或因特网上不同位置的多名开发人员在访问和操作同一个源代码库。但是，CVS 最大的优点还在于多名开发人员能同时在一个相同的文件上工作。而 RCS 通过其加锁机制，一个时刻只能允许一个人编辑某个文件，CVS 却没有这样的要求，它会试着把几个开发人员对同一文件的修改合并到一起。如果遇到不能解决的冲突，需要进行手工干预。下面的章节将会更详细地讨论 CVS 的这些特性以及其他一些特性。

7.3.2 设置 CVS

在设置 CVS 源代码库之前首先要创建这个源代码库。创建源代码库先要决定库的位置，然后用 cvs init 命令初始化该库。出于讨论的目的，我们把源代码库建在\$HOME/cvs 目录下：

```
$cvs -d $HOME/cvs init
```

-d 指定了要初始化的 CVS 源代码库。init 命令创建目录并且把一系列用于管理源代码库的文件存放到子目录 CVSROOT 下。绝对不要直接编辑这些文件；使用 CVS 命令去操作它们，否则会让你的源代码库变得无法使用。

一旦该目录的初始化工作完成，立即设置环境变量 \$CVSROOT，把它指向这个目录。如果正在使用 bash 或其他 Bourne shell 的变体，执行 `export CVSROOT=$HOME/cvs.C shell` 及其变体应该使用 `setenv CVSROOT /usr/src/repos`。为了方便，可以把这条语句放到 shell 的初始化文件中，这样在用户每次登录时 \$CVSROOT 变量就设置好了。

提示： 一个源代码库能包含多个项目。

下一步是将你的源代码文件（你的项目）交给 CVS 控制。最简单的做法是使用 CVS 的 import 命令。import 的语法是：

```
cvs import [ -d ] [ -k subst ] [ -I ign ] [ -m msg ] [ -b branch ] [ -W
spec ] repository vendor-tag release-tags...
```

提示： 大多数 CVS 命令都以 cvs 为前缀。

表 7.4 介绍了 import 命令的选项。

表 7.4 CVS import 命令的选项和参数

域	描述
-d	用每个导入文件最后修改的时间作为 CVS 导入时间
-k sub	设置 RCS 关键字的默认替代模式
-I ign	忽略文件列表
-b bra	指定开发者的分支 ID
-m msg	记录导入时的消息
repository	从源代码库 repository 中导入文件
vendor-tag	是源代码提供者的名字
release-tags	指定用于某个特殊发布的符号名

假定你要装入 CVS 的文件位于 /home/joebob/src，而你想让它们出现在 src 目录下的源代码库。

```
$cd /home/joebob/src
$cvs import src yoyo start
```

下面的例子把保存在 \$HOME/src 目录下本章的源代码文件 yo.c 和 prn_env.c 的副本导入位于 \$CVSROOT/chap07 目录下的源代码库。在本例中，vendor-tag 和 release-tags 是占位符，但却不是必须要有的。它们只是在把第三方源代码导入源代码库时才有真实的含义。

```
$ cd src
$ cvs import -m "Initial check in of imported files" chap07 lpu2 start
N chap07/prn_env.c
N chap07/yo.c
```

```
No conflicts created by this import
$
```

如果不指定 `-m msg`, 就会进入 `vi` 会话并且提示输入信息。CVS 源代码库的列表表明文件已经被成功地检入了:

```
$ ls $CVSROOT/chap07
total 2
-r--r--r--      1  kwall  users   686 Aug 13 16:27 prn_env.c,v
-r--r--r--      1  kwall  users   552 Aug 13 16:27 yo.c,v
$
```

要证实源代码库能工作, 可以把原来的目录换个名字, 再把源代码检出, 用 `diff` 比较原来的目录和新目录 (下一节将介绍如何检出源代码) 之间差别:

```
$ cd ..
$ mv src src.orig
$ cvs checkout chap07
cvs checkout: Updating chap07
U chap07/prn_env.c
U chap07/yo.c
$ diff -r src.orig chap07
Only in chap07: CVS
$ rm -rf src.orig
```

两个目录之间惟一的区别是 CVS 目录是由 `checkout` 命令创建的, 这表明该命令执行成功。把原来的源代码目录改名可以避免意外地编辑那些文件: 把文件检入 CVS 后, 用户应该只编辑从源代码库检出的文件。

就本章而言, 上面就是需要的全部管理性工作。本章的其余部分介绍 CVS 的一般用法。此时, 你的源代码树已经处于 CVS 的控制之下了。所以可能你想知道怎样访问它——读取。

7.3.3 检出源代码文件

要使用保存在 CVS 源代码库中的文件, 必须把它们检出, 这和 RCS 的情形是一样的。但是, 和 RCS 不同, 你通常使用的是一棵完整的源代码树, 它对应于某个项目。所以继续使用前面的例子检出项目 `chap07`, 发出 `checkout` 命令:

```
$ cvs checkout chap07
cvs checkout: Updating chap07
U chap07/prn_env.c
U chap07/yo.c
$ cd chap07
```

除非用 `-d` 选项指定目录, 否则 CVS 使用 `$CVSROOT` 变量来确定源代码库的位置。`chap07` 目录下的大多数文件都是源代码文件。但是 CVS 子目录包含了 CVS 用于跟踪 `chap07` 变化的其他文件。

从源代码库检出文件后, 可以在检出的文件上工作。惟一要说明的是当你对代码做过修改而要保存时, 必须把修改过的文件检入源代码库。这会在下一节介绍。

7.3.4 将改动合并进源代码库

在你修改了工作目录下的文件后，这种改动只有你自己看得见。把改动合并进源代码库要分两步走。首先，你必须确保其他开发人员对你正在编辑的文件所做的修改也已经反映到你的源代码上了。这叫作让你的工作文件和源代码库同步（sync）。接下来，把改动提交给源代码库。

假定你已经编辑好了 `yo.c`。要把工作目录同源代码库进行同步操作，需在你的工作目录下执行 `update` 命令。

```
$ cvs update
cvs update: Updating.
RCS file: /home/kwal/cvs/yo.c,v
retrieving revision 1.2
retrieving revision 1.3
Merging differences between 1.2 and 1.3 into yo.c
M yo.c
```

这里显示地表明有人改变了 `yo.c`。CVS 检测到了这一点，并检索出了有关的版本，然后尝试进行合并。最后一行 `M yo.c` 意味着你所做的修改其他人还看不到。看过合并进 1.2 和 1.3 版本对 `yo.c` 的修改后，你就可以使用 `commit` 命令提交你自己的修改：

```
$ cvs commit yo.c
Checking in yo.c
/home/kwall/cvs/chap07/yo.c,v <-- yo.c
new revision: 1.4; previous revision: 1.3
done
```

下面将会进入 `vi` 会话，提示输入一条日志信息。此时，你所做的修改其他程序员已经能看到了。

7.3.5 检查改动

要检查一个文件的修改历史，可使用 `log` 命令：

```
$ cvs log yo.c

RCS file: /home/kwall/cvs/chap07/yo.c,v
Working file: yo.c
head: 1.6
branch:
locks: strict
access list:
symbolic names:
    start: 1.1.1.1
    lpu2: 1.1.1
keyword substitution: kv
total revisions: 7;      selected revisions: 7
description:
-----
```

```
revision 1.6
date: 2000/08/13 23:47:43; author: kwall; state: Exp; lines: +0 -1
Last change before release.
-----
revision 1.5
date: 2000/08/13 23:44:38; author: kwall; state: Exp; lines: +1 -0
Added blank line before the return statement.
-----
revision 1.4
date: 2000/08/13 23:43:41; author: gomer; state: Exp; lines: +1 -1
Changed the return statement, again.
```

输出的关键信息是在第一行连字符后面的内容。它显示出和 `rlog` 命令类型相同的日志信息：一个修订版本号，后面跟着日期和时间戳以及其他相关信息，再后面是每个修订版本在检入时输入的日志信息。

7.3.6 添加和删除文件

除了和 RCS 类似的行为，CVS 还维护了工作目录的历史，用户的工作文件就处在这个目录之下。但是，它不会自动假定一个被删除的文件代表该文件也应从源代码库删除，或者一个新文件也不代表该文件应该添加到源代码库中。你也可以通过把 `commit` 命令和 `add` 和 `remove` 命令联合使用来进行控制。

要把一个文件加入源代码库：

1. 创建该文件。
2. 用 `add` 命令加入。
3. 用 `commit` 命令提交给源代码库。

下面的清单演示了如何把文件 `yo.h` 加入源代码库：

```
$ cvs add yo.h
cvs add: scheduling file 'yo.h' for addition
cvs add: use 'cvs commit' to add this file permanently
$ cvs commit -m "Added header file" yo.h
RCS file: /home/kwall/cvs/chap07/yo.h,v
done
Checking in yo.h;
/home/kwall/cvs/chap07/yo.h,v <-- yo.h
initial revision: 1.1
done
$
```

第一条命令把 `yo.h` 加入源代码库。但是，在发出 `commit` 命令之前它都不会真正移入源代码库。`commit` 命令使用 `-m` 选项而不是从提示行输入来指定一条日志信息。`RCS file: /home/kwall/cvs/chap07/yo.h,v` 一行表明 CVS 是以 RCS 为基础的——CVS 实际上使用了 RCS 命令把文件加入源代码库并且维护源代码库。

从源代码库删除一个文件也遵循类似的步骤：

1. 从源代码库删除该文件或改名。
2. 对该文件名执行 `cvsm rm` 命令。
3. 使用 `commit` 命令提交删除操作。

下面的例子说明了如何把文件 `main.c` 从源代码库中删除：

```
$ rm main.c
$ cvs rm main.c
cvs remove: scheduling 'main.c' for removal
cvs remove: use 'cvs commit' to remove this file permanently
$ cvs commit
cvs commit: Examining .
Removing main.c;
/home/kwall/cvs/chap07/main.c,v <-- main.c
new revision: delete; previous revision: 1.1
done
$
```

即便文件已经从源代码库中删除，它还是会在源代码库中留下日志历史。虽然本例从工作目录删除了该文件，但只是简单地把文件改名，然后用 `commit` 命令提交原来的文件名，这会给你更多的选择机会。如果你决定确实需要保留那个文件，那这样做能让你有能力改变自己主意。

7.3.7 解决文件冲突

不可避免地，CVS 不能解决由多人对同一文件所做的多次编辑造成的冲突。假如程序员 `sue` 也正在 `yo.c` 上工作。当她更新自己的源代码库时，她得到了如下输出：

```
$ cvs update
cvs update: Updating.
RCS file: /home/kwall/cvs/chap07/yo.c,v
retrieving revision 1.5
retrieving revision 1.7
Merging differences between 1.5 and 1.7 into yo.c
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in yo.c
C yo.c
U yo.h
$
```

CVS 没有打印出 `M yo.c` 来表示已经合并了差异，而是显示 `C yo.c` 说明出现了无法解决的冲突。`U yo.h` 消息指出在源代码库中出现了一个新文件，而 `sue` 的工作目录用它做了更新。

为了解决冲突，用编辑器打开该文件。它的内容如下：

```
/*
 * yo.c - Code to demonstrate RCS usage
 */
```


表 7.6 CVS 命令行选项和参数

选项/参数	描述
-d cvsroot	使用 cvsroot 而不是\$CVSROOT 作为源代码库的根目录
-e editor	使用 editor 编辑日志信息
-f	不使用 ~/.cvsrc 文件
-H	显示 cvs 用法信息
-l	关闭历史日志
-n	不执行任何修改磁盘的命令
-q	取消一些 CVS 输出
-Q	取消大多数 CVS 输出
-r	让检出的文件只读(默认情况为可读写)
-s var=val	设置 CVS 用户变量 var 的值为 val
-T tmpdir	使用 tmpdir 目录保存临时文件
-v	CVS 版本和版权

要了解有关 CVS 使用的更多信息, 参考如下资源:

- <http://www.cvshome.org/>
- CVS 的 info 页面, info cvs
- CVS 的手册页面, cvs(1)和 cvs(5)

7.4 小 结

在这一章, 你学习了修订控制系统 (Revision Control System, RCS) 和并发版本系统 (Concurrent Version System, CVS)。ci 和 co 及其多种选项和参数是 RCS 的基本命令。RCS 关键字能够让你向代码和编译后的程序嵌入识别信息, 这些信息以后可以用 ident 命令提取出来。你还学习了其他有用但不常用到的 RCS 命令, 包括 rcsdiff、rcsclean、rcsmerge 和 rlog。CVS 给 RCS 的基础命令增加了更加丰富、更面向于项目的接口, 从而让多个程序员可以在同一文件上工作, 而且能够在一个中央放置的服务器上的单个源代码库中保存多个项目。

第 8 章 调 试

虽然我们非常不愿意承认，但是软件中还是会有错误存在。本章描述自由软件联盟（Free Software Foundation, FSF）的主要软件工具之一，GDB，即 GNU DeBugger。虽然调试工作一点也不能随心所欲，但是 GDB 的高级特性可以减轻你的负担，提高工作效率。如果你可以跟踪代码并且能在几分钟内修正一个严重错误的话，那么花在学习 GDB 上的时间和精力还是值得的。

8.1 为使用 GDB 进行编译

正如你在第 3 章学到的那样，对代码进行调试要求你的源代码在编译时用 -g 选项，以生成增强的符号表。因此，下面的命令：

```
$gcc -g file1.c file2.c -o prog
```

将使得 prog 和调试符号一起在它的符号表里被创建。如果你愿意的话，可以使用 gcc 的 -ggdb 选项生成更多的（GDB 特有的）调试信息。但是，为了更有效地工作，这个选项要求你可以访问你所链接的各个库的源代码。虽然这在某些情况下非常有用，但是在磁盘空间上花费较多。在大多数情况下，普通的 -g 选项就可以了。

同样在第 3 章里提到过，可以同时使用 -g 和 -O(优化)选项。但是，因为优化改变了结果程序，你所期待的程序代码和可执行的二进制代码之间的关系就可能不复存在了。原来程序里的变量或者代码行可能会不见踪影，而原来程序里面没有的变量赋值会在你没有料到的地方突然冒出来。我的建议是直到你已经彻底将代码调试完毕后才开始进行代码优化工作。从长远来看，这将使你的编程工作，特别是你用于调试代码的那部分工作变得简单而轻松。

警告： 如果你的程序是以二进制的形式分发的，那么不要从你的二进制代码中去除调试符号。因为这不仅是对你的用户的礼貌，而且也会对你自己有益。如果你从一个用户那里得到一个关于程序错误的报告（那个用户得到的只是程序的二进制版本），如果你为了使二进制代码变得短小一些而用 strip 命令除去了所有的符号，那么她将不能提供有用的信息。虽然她可能愿意去下载源代码以便开启调试选项再把程序编译一遍，找到错误所在，但是你这样要求用户未免不太近人情。如果你正在一个专业环境中工作，可能会要求你去除二进制代码中的调试符号以保证更难对代码做逆向回溯。

8.2 使用基本的 GDB 命令

需要用 GDB 完成的大部分工作都可以用很少的命令集合完成。本节教你一些刚好够用的 GDB 命令。8.3 节将讨论一些方便好用的高级功能。

8.2.1 启动 GDB

要启动调试会话，简单地输入 `gdb progname [corefile]` 即可，用你想调试的程序名字替换 `progname`。使用 `core` 文件是可选的，但能增强 GDB 的调试能力。本章大多数例子都使用了程序清单 8.1 给出的程序。

程序清单 8.1 一个有错误的程序

```
/*
 * debugme.c - Poorly written program to debug
 */
#include <stdio.h>
#include <stdlib.h>

#define BIGNUM 5000

void index_to_the_moon(int ary[]);

int main(void)
{
    int intary[100];

    index_to_the_moon(intary);

    exit(EXIT_SUCCESS);
}

void index_to_the_moon(int ary[])
{
    int i;
    for(i = 0; i < BIGNUM; ++i)
        ary[i] = i;
}
```

用 `make debugme` 命令编译这个程序。

在大多数系统上，如果试着通过键入 `./debugme` 来执行该程序，它都会立即产生一个段错误（segmentation fault）并转储内存（dump core）。

第一步是用该程序名 `debugme` 和内存转储文件 `core` 作为参数启动 GDB：

```
$ gdb debugme core
```

在完成 GDB 初始化后，屏幕应该出现类似图 8.1 的内容：

```
$ gdb debugme core
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-COL-linux"...
Core was generated by './debugme'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Reading symbols from /lib/ld-linux.so.2...done.
#0  index_to_the_moon (ary=0x7ffffca0) at debugme.c:24
24                                ary[1] = 1;
(gdb) █
```

图 8.1 GDB 启动屏幕

如果你不喜欢许可信息（我讨厌它们），使用-q（或者--quiet）选项可以不显示它们。另一个有用的命令行选项是-d dirname，dirname是个目录名，该目录告诉GDB到哪里去找源代码（默认在当前工作目录下查找）。正如你在图中所看到的那样，GDB报告生成core文件的可执行文件名以及程序终止的原因。在这个例子里，程序产生了一个信号11，它代表段错误。GDB还能显示出它正在执行的函数以及它认为导致出错的程序行（第24行），这些对调试程序都很有用。

你想做的第一件事是在调试环境下运行这个程序。做这件事情的命令是run。你的程序在通常情况下可以接受的任何参数都可以作为run命令的参数传入。另外，程序会接受一个正确建立的shell环境，该环境由环境变量\$SHELL的值来决定。但是如果你愿意，在启动了调试会话后可以使用GDB的set和unset命令来分别设置或取消参数和环境变量。具体做法为，键入set args arg1 arg2，这里的arg1和arg2（或者任何数目的参数）是被调试的程序需要的选项和参数。使用set environment env1 env2来设置环境变量（同样，env1和env2代表了准备设置或取消的环境变量）。

提示： 如果你忘记了一条GDB命令或者不能肯定它的确切语法，GDB提供了丰富的帮助系统。在GDB提示符下键入不带参数的help命令可以得到一个简短的命令清单，但是help topic将打印该主题(topic)的帮助信息。GDB总带有一个完整的TeXinfo格式的帮助系统以及一本优秀的手册《Debugging with GDB》，它们可以联机获得或者从自由软件联盟那里用邮件订单获得。

尝试在调试器里运行这个程序，GDB在收到信号SIGSEGV后停止运行：

```
(gdb) run
Starting program: /home/kwall/lpu2/08/debugme
Program received signal SIGSEGV, Segmentation fault.
```

```
index_to_the_moon (arg=0x7ffffc90) at debugme.c:24
24             ary[i] = i;
(gdb)
```

这一简短的输出清单表明段错误出现在 debugme.c 中第 24 行的函数 index_to_the_moon 里。注意 GDB 能够显示出错误代码的行号。它还能说明错误出现的地址，0x7ffffc90，这也是潜在的有用信息。

8.2.2 在调试器中查看代码

当然问题是：函数 index_to_the_moon 究竟出了什么问题？你可以执行 backtrace 命令以生成导致段错误的函数树。这个功能追踪（在我的系统上针对我的程序）出来的结果如下：

```
(gdb) backtrace
#0 index_to_the_moon (ary=0x7ffffc90) at debugme.c:24
#1 0x80483df in main () at debugme.c:15
(gdb)
```

提示：使用 GDB 时不必键入完整的命令名称。任何惟一的缩写都可以用。例如，要使用 backtrace，键入 back 就足够了。

所以，问题实际上出现在 main 函数调用 index_to_the_moon 函数的地方，GDB 显示是在 debugme.c 中第 15 行。

但是，知道代码中出错行的上下文对调试程序是有帮助的。为了达到这个目的，可使用 list 命令，该命令通常的形式为 list [m,n]。m 和 n 是要显示的包含错误首次出现位置的代码段的起始行和结尾行的行号，不带参数的 list 命令将显示附近的 10 行代码，显示如下：

```
(gdb) list
19
20 void index_to_the_moon(int arg[])
21 {
22     int i;
23     for (i = 0; i < BIGNUM; ++i)
24         arg[i] = i;
25 }
(gdb)
```

清楚地看到代码中在什么地方进行了什么操作，你就能判断出问题出在哪儿，并且解决它。

8.2.3 检查数据

GDB 最有用的特性之一是它能够显示被调试程序中几乎任何表达式、变量或者数组的类型和值。它能以编写程序所用的语言打印出任何合法表达式的值。完成此项任务的命令，不出所料，是 print。下面是一些 print 命令和它们的输出结果：

```
(gdb) print i
$1=4999
(gdb) print ary[i]
Cannot access memory at address 0x80004aac.
```

这个例子继续了早先调试 `debugme.c` 的那个例子：你还要尝试找出 `debugme` 崩溃的原因和位置。虽然在这个例子中，程序崩溃时变量 `i` 的值累计到 4999，但是它在你的系统上崩溃的位置可能取决于系统内存的格局、处理器内存空间和系统中可用内存的数量等因素。

第二条命令 `print ary[i]` 明确说明程序不能访问指定的内存，即使它确实能够访问前一个变量。`$1` 和 `$2` 是被检查数值的历史记录项。如果你想在以后访问这些值，就可以使用这些别名而不用重新键入命令。例如，命令 `$1-1` 产生

```
(gdb) print $1-1
$2=4998
```

你并没有受到限制只能使用离散的内存地址值，因为 GDB 可以显示一个指定内存区域的值。要打印从 `ary` 开始的头 10 个内存区域，可以使用下面的命令：

```
(gdb) print ary@10
$ 3 = {0x7ffffc90, 0x7ffffcb8, 0x2aab5930, 0x804957c, 0x804957c,
0x2aba3fd1, 0x8049488, 0x7ffffcb8, 0x80483bb, 0x8049474}
```

`@10` 表示打印从 `ary` 开始的 10 个值。另一方面，如果你想打印从 `ary` 的第 1 个元素开始的 5 个数组元素数值。可以用如下命令：

```
(gdb) print ary[1]@5
$4 = {1, 2, 3, 4, 5}
(gdb)
```

因为每个 `print` 命令都创建一个 GDB 命令历史记录项，所以你可以在以后成组使用数组值。

你可能不明白为什么第一个 `print` 命令显示的是十六进制的值而第二个 `print` 命令显示的则是十进制的值。首先，记住 C 语言里的数组下标是从 0 开始的，还要记住空数组名是一个指向数组起点的指针。因此，GDB 发现 `ary` 是数组的基地址，然后按内存地址的方式显示它以及后面的 9 个值。内存地址习惯上以十六进制数显示。如果你想用十进制方式显示 `ary` 的前 10 个值，使用下标符 `[]` 和第一个数值的下标 0，如下所示：

```
(gdb) print ary[0]@10
$5 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

注意： GDB 通常在支持 GNU 的 `readline` 库的情况下被编译，这意味着 `gdb` 支持 `bash shell` 的命令行编辑和历史记录特性。例如，回忆以前的命令，可使用上箭头回卷命令历史记录。参看 `readline` 手册页获得更多的关于命令行编辑的信息。

GDB 还可以用 `whatis` 命令告诉你变量的类型：

```
(gdb) whatis i
type = int
(gdb) whatis ary
type = int *
(gdb) whatis index_to_the_moon
type = void (int *)
```

这个特性可能看起来非常没用，因为你当然知道程序中所有变量的类型（是的，正是这样！）。但是，当你第一次要调试别人的代码或者修正一个很久没看过的，有多个文件的项目时，你将改变你的观点。

8.2.4 设置断点

当你调试出问题的代码时，在某一点停止执行往往很管用。GDB 允许你在几种不同的代码结构上设置断点，包括行号和函数名，还允许你设置条件断点，如果一定的条件被满足，代码将停止执行。

要根据行号设置断点，使用如下语句：

```
(gdb) break linenum
```

要根据函数名设置断点，则使用：

```
(gdb) break funcname
```

在以上两种情况中，GDB 将在执行指定的行号或进入指定的函数之前停止执行程序。你可以使用 `print` 显示变量的值，或者使用 `list` 查看将要执行的代码。如果你有一个多文件项目，且想在执行到非当前源文件的某行或某函数时停止执行，可以使用如下形式的命令：

```
(gdb) break filename:linenum
(gdb) break filename:funcname
```

条件断点通常更有用。它们允许你在一定条件满足时临时停止程序的执行。设置条件断点的正确语法是：

```
(gdb) break linenum if expr
(gdb) break funcname if expr
```

表达式可以是真值表达式（非 0）。例如，下面的断点命令在 `debugme` 程序的第 25 行停止执行，这时 `i` 等于 15：

```
(gdb) break 24 if i==15
Breakpoint 1 at 0x8048410: file debugme.c, line 24.
(gdb) run
Starting program: /home/kwall1/lpu2/08/debugme
Breakpoint 1, index_to_the_moon (ary=0x7ffffb28) at debugme.c:24
24          ary[i] = i;
```

正如你所看到的，GDB 在第 24 行停止。用 `print` 命令可以快速地确认程序是按我们的要求在 `i` 等于 15 时停止执行的：


```
(gdb) print i
$2 = 15
```

当你键入 `run` 命令，程序已经开始执行，GDB 会告诉你程序已经启动，然后问你是否想从程序起点重新开始，选择 `Yes`。

选择是要继续断点以后的程序执行，简单地键入 `continue`。如果设置了很多断点，却忘了到底设置了哪些以及哪些断点已经被触发，可以使用 `info breakpoints` 命令来更新你的记忆。`delete` 命令允许删除断点，或者可以简单地使断点无效。图 8.2 描述了下面这些命令的输出：

```
(gdb) info breakpoints
(gdb) delete 1
(gdb) disable 2
(gdb) info breakpoints
```

```
(gdb) info breakpoints
Num Type      Disp Enb Address      What
1  breakpoint keep y  0x08048410 in index_to_the_moon at debugme.c:24
   stop only if i == 15
   breakpoint already hit 1 time
2  breakpoint keep y  0x08048410 in index_to_the_moon at debugme.c:24
   stop only if i > 25
(gdb) delete 1
(gdb) disable 2
(gdb) info breakpoints
Num Type      Disp Enb Address      What
2  breakpoint keep n  0x08048410 in index_to_the_moon at debugme.c:24
   stop only if i > 25
(gdb) █
```

图 8.2 gdb 有着管理断点的复杂功能

在图 8.2 里，使用 `info breakpoints` 命令来获得断点列表，删除第 1 个断点，使第 2 个断点无效，然后再显示断点信息。使一个被置为无效的断点重新有效的命令是 `enable N`，`N` 是断点号。

8.2.5 检查并更改运行中的代码

你已经见过 `print` 和 `whatis` 命令了，因此不再重复它们，除了要指出如果你使用 `print` 命令显示一个表达式的值，而这个表达式改变了程序使用的变量，那么就改变了一个正在运行中的程序的变量值。这可能不是什么坏事，但是需要知道你所做的事情有副作用。

`whatis` 命令的一个缺点是它只给你一个变量或者函数的类型。如果你有一个结构如下所示：

```
struct s {  
    int index;  
    char *name  
} *S;
```

然后键入:

```
(gdb) whatis S
```

GDB 只报告结构的名称:

```
type = struct s *
```

如果你想得到结构的定义, 可以像下面那样使用 `ptype` 命令:

```
(gdb) ptype s  
type = struct s {  
    int index;  
    char *name;  
} *
```

如果你想改变一个变量的值 (记住这种改变将影响正在运行的代码), 使用 GDB 命令是 `set`, 相应的代码如下:

```
(gdb) set variable varname = value
```

`varname` 是你想改变的变量, `value` 是 `varname` 的新值。回到程序清单 8.1, 使用 `delete` 命令删除所有你已设置的断点和观察点, 然后设置如下断点:

```
(gdb) break 24 if i == 15
```

然后运行程序, 当 `i` 等于 15 时, `gdb` 将临时中止程序执行。在遇到这个断点后, 给 GDB 以下命令:

```
(gdb) set variable i = 10
```

这将 `i` 的值重新设置为 10。执行一次 `print i` 命令以确认变量值已经被重置, 然后执行 3 次 `step` 命令, 再执行一次 `print i` 命令。你将看到 `i` 的值随着 `for` 循环一次而增 1。除了演示你可以改变一个正在执行的程序, 这些 `step` 命令也展示了怎样在程序中进行单步跟踪。`step` 命令一次执行程序中的一个语句。

注意: 不需要键入 `step` 三次。GDB 记住了最后一个被执行的命令, 因此你可以简单的按 Enter 键来重复执行最后的命令, 从而减少了键盘输入。这对大多数 GDB 命令有效。参见文档以获得更多信息。

当遇到一个函数的时候, `next` 命令执行整个函数, 而 `step` 命令将只单步进入函数, 每次仍然继续执行一个语句。

最后我将讨论的对一个正在运行的程序进行检查的方法是使用 `call`、`finish` 以及 `return` 命令来调用函数。表 8.1 列出了执行命令的语法和功能。

表 8.1 用于函数的命令

命令	描述
call name(args)	调用并执行名为 name, 参数为 args 的函数
finish	如果可以, 则中止当前函数并打印它的返回值
return value	停止执行当前函数, 并将 value 返回给调用者

call 命令的使用和你调用某个有名字的函数完全相同。例如, 命令

```
(gdb) call index_to_the_moon(intary)
```

执行参数为 intary 的函数 index_to_the_moon。你可以像普通情况下调用函数那样对函数设置断点并使用 GDB 的各种功能。如果你在函数内部设置断点, 可以用 continue 来恢复执行, 用 finish 来完成函数调用以及在函数返回时停止执行, 或者使用 return 命令退出函数, 并且在函数正常返回前向调用者返回一个特定值。你还可以使用 return 命令返回一个随意值来测试边界条件。

8.3 高级 GDB 概念和命令

本小节讨论一些复杂的概念和命令, 它们将使你更有效、更成功地使用 GDB。要讨论的概念包括 GDB 的变量作用域和上下文的概念。要讨论的命令包括遍历函数调用栈、操纵源文件、与 Shell 进行通信以及将 GDB 粘附到某个已存在的进程。

8.3.1 变量的作用域和上下文

变量的上下文是指变量是活动的还是不活动的。如果一个变量能被访问或操作则是活动的。相反, 如果一个变量不能被访问或操作则是不活动的, 或者说超出了上下文 (out of context)。有几条规则定义了构成活动变量或者上下文中的变量的是什么。

- 如果某个函数 (称作控制函数) 正在执行或者该函数将控制流程传给一个由控制函数调用的函数, 那么该函数里的局部变量处于活动状态。例如函数 foo 调用函数 bar; 只要 bar 在执行, 所有 foo 和 bar 的局部变量就是活动的。一旦 bar 返回, 就只有 foo 函数的局部变量是活动的, 从而也只有 foo 函数的局部变量能够被访问。
- 全局变量不管程序是否运行总是活动的。
- 非全局变量是非活动的, 除非程序运行。

关于活动变量就这么多。那么变量上下文的概念又是什么呢? 这个概念的复杂性在于静态变量的使用, 静态变量对文件是局部的。那就是说, 你可以在多个文件中使用相同名字的静态变量, 它们相互之间不会发生冲突, 因为静态变量在定义它们的文件外是不可见的。幸运的是, GDB 有办法标识你指的是哪个变量。这和 C++ 里面的域解析操作符相似。语法是:

```
file::varname
funcname::varname
```

`varname` 是你想要引用的变量名, `file` 和 `funcname` 是含有 `varname` 变量的文件或者函数的名称。例如, 给定两个源代码文件 `foo.c` 和 `bar.c`, 每个文件都含有一个名为 `baz` 的变量, `baz` 被声明为静态变量。要用 `foo.c` 文件里的 `baz` 变量, 你可以像下面这样写:

```
(gdb) print 'foo.c'::baz
```

将文件名扩起来的单引号是必须的, 以便 GDB 知道你指的是一个文件名。同样, 给定两个函数, `blat` 和 `splat`, 每个函数都有一个整数变量 `idx`, 下面的命令将打印 `idx` 变量在这两个函数中的地址:

```
(gdb) print &blat::idx
(gdb) print &splat::idx
```

8.3.2 遍历函数堆栈

GDB 提供两个命令在调用栈中进行上下移动, 调用栈是使你到达当前代码位置的函数调用链。假想一个程序, 程序里 `main` 函数调用 `make_key` 函数, `make_key` 函数调用 `get_key_num` 函数, `get_key_num` 函数调用 `number` 函数。程序清单 8.2 说明这种情况。

程序清单 8.2 一个有 3 个函数调用深度的调用链

```
/*
 * callstk.c - Illustrate traversing the call stack with GDB
 */
#include <stdio.h>
#include <stdlib.h>

int make_key(void);
int get_key_num(void);
int number(void);

int main(void)
{
    int ret = make_key();
    printf("make_key returns %d\n", ret);
    exit(EXIT_SUCCESS);
}

int make_key(void)
{
    int ret = get_key_num();
    return ret;
}

int get_key_num(void)
{
    int ret = number();
    return ret;
}
```

```

    }

    int number(void)
    {
        return 10;
    }

```

使用断点命令，在第 31 行设置断点：

```

(gdb) break 31
Breakpoint 1 at 0x80484fc: file callstk.c, line 32.

```

接着用 `run` 命令运行程序。程序在第 31 行停止。然后，键入命令 `where`，它将打印如下结果（你机器上显示的十六进制地址很可能会有所不同）。

```

(gdb) where
#0 number () at callstk.c:31
#1 0x80484cf in make_key () at callstk.c:20
#2 0x804849b in main () at callstk.c:13
(gdb)

```

显示出的是逆序的调用链，就是说，`number` 函数（#0 断点）被 `make_key` 函数（#1 断点）调用，`make_key` 函数被 `main` 函数调用。`up` 命令将调用栈上移一个函数调用，移到 20 行。`down` 命令将控制移回 `number` 函数。执行 `step` 命令 3 次使 `number` 函数返回到调用它的 `get_key_num` 函数，这时你可以打印变量 `ret`，看它的值。图 8.3 显示了刚才描述的命令序列的结果。

```

(gdb) break 30
Breakpoint 1 at 0x8048484: file callstk.c, line 30.
(gdb) run
Starting program: /home/kwall/lpu2/08/callstk

Breakpoint 1, number () at callstk.c:31
31      {
(gdb) where
#0 number () at callstk.c:31
#1 0x804844f in make_key () at callstk.c:20
#2 0x804841b in main () at callstk.c:13
(gdb) up
#1 0x804844f in make_key () at callstk.c:20
20          int ret = get_key_num();
(gdb) down
#0 number () at callstk.c:31
31      {
(gdb) step
number () at callstk.c:32
32          return 10;
(gdb) step
0x8048490      33      }
(gdb) step
get_key_num () at callstk.c:27
27          return ret;
(gdb) print ret
$1 = 10
(gdb)

```

图 8.3 遍历 callstk 程序的调用链

遍历调用链的命令和检查变量值的命令结合起来,使你能够从底层观察一个正在运行的程序的详细情况。这种底层的控制对你提高调试技巧和追踪隐藏在调用链中的错误将是无价之宝。

8.3.3 操纵源代码文件

在多个文件的项目中定位一个指定函数对于 GDB 来说是雕虫小技。只要你像 8.2.1 节中提到的那样用 `-d` 开关告诉 GDB 到哪里去找其他的源代码。当所有的源文件代码不在当前工作目录或程序的编译目录(这个目录记录在程序的符号表里)下时,这个选项开关特别有用。要指定一个或多个其他目录,可以用一个或多个 `-d <路径名>` 选项启动 GDB,如下所示:

```
$ gdb -d /source/project1 -d /oldsource/project1 -d /home/gomer  
/src killerapp
```

要定位一个特定字符串在当前文件中的下一次出现,可以使用 `search <字符串>` 命令。使用反向查找命令 `reverse-search<字符串>` 来查找字符串上一次出现的地方。执行程序清单 8.2, 假定程序在前面例子中设置的断点第 31 行处停下来。如果你想查找“`return`”这个词上一次出现的地方,使用如下命令:

```
(gdb) reverse-search return
```

GDB 去查找然后显示如下文本信息:

```
(gdb) reverse-search return  
29                return ret;
```

在有数百行的大型程序中使用 `search` 和 `reverse-search` 命令非常管用。

8.3.4 与 Shell 进行通信

当运行一个调试会话时,你会经常需要在 shell 命令提示符下执行命令。GDB 提供 `shell` 命令使你不用离开 GDB 就能执行 shell 命令。这个命令的语法是:

```
(gdb) shell command
```

`command` 是你想执行的 shell 命令。假定你已经忘了当前的工作目录,可以简单地执行下面的命令:

```
(gdb) shell pwd
```

GDB 将把命令 `pwd` 传给 shell, 可在 `/bin/sh` 中执行:

```
/home/kwall/projects/lpu2/08/src
```

8.3.5 附加到某个运行中的程序

最后,我们要讨论的高级特性是怎样使用 GDB 附加到一个正在运行的进程上,比如一个系统守护进程。完成这项任务的过程和启动 GDB 的过程非常相似,只不过传递给 GDB 的第二个参数是你要附加的可执行程序的 PID (进程 ID),而不是内存转储文件 (core) 的

名字。GDB 会抱怨说没有发现名为 PID 的程序，你无须理会它，因为 GDB 总是首先检查文件是否有叫这个名字的内存转储文件。图 8.4 是使用命令 `gdb /usr/sbin/httpd 728` 试图将 GDB 附加到我系统的 httpd 服务器守护进程上的截断显示。注意，如果程序不在当前目录下，则必须指定正确的路径。

```
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-COL-linux"...(no debugging symbols found)...

/home/kwall/lpu2/08/728: No such file or directory.
Attaching to program: /usr/sbin/httpd, process 728
Reading symbols from /lib/libm.so.6...done.
Reading symbols from /lib/libcrypt.so.1...done.
Reading symbols from /lib/libdb.so.3...done.
Reading symbols from /lib/libdl.so.2...done.
Reading symbols from /lib/libc.so.6...done.
Reading symbols from /lib/ld-linux.so.2...done.
Reading symbols from /lib/libnss_nis.so.2...done.
Reading symbols from /lib/libnsl.so.1...done.
Reading symbols from /lib/libnss_files.so.2...done.
Reading symbols from /lib/libnss_compat.so.2...done.
Reading symbols from /usr/libexec/apache/mod_vhost_alias.so...done.
Reading symbols from /usr/libexec/apache/mod_env.so...done.
Reading symbols from /usr/libexec/apache/mod_log_config.so...done.
Reading symbols from /usr/libexec/apache/mod_mime_magic.so...done.
Reading symbols from /usr/libexec/apache/mod_mime.so...done.
Reading symbols from /usr/libexec/apache/mod_negotiation.so...done.
Reading symbols from /usr/libexec/apache/mod_status.so...done.
Reading symbols from /usr/libexec/apache/mod_info.so...done.
Reading symbols from /usr/libexec/apache/mod_include.so...done.
Reading symbols from /usr/libexec/apache/mod_autoindex.so...done.
---Type <return> to continue, or q <return> to quit---
```

图 8.4 附加到 httpd 服务器守护进程上

提示： 为了把 GDB 附加到一个正在运行的程序上，必须能够访问它的进程空间。例如，一个普通用户除非 `su` 成超级用户，否则不能把 GDB 附加到 httpd 守护进程上。

附加一个正在运行的进程将会自动将其停止以便你能使用常规的 GDB 命令检查它的状态。你可以使用 `top` 或 `ps` 命令检查程序的状态来证实该程序已被停止。要从进程上分离出来可使用 `detach` 命令，或者如果你想分离并退出 GDB，则键入 `quit`，该命令能够从运行的进程上分离并允许其继续执行（当然是退出 GDB）。

如果你已经在允许 GDB 了，可以使用 `attach` 和 `file` 命令来附加到一个运行的程序上。首先，使用 `file` 命令指定运行在进程中的程序名，而要加载它的符号表，可以按如下操作：

```
(gdb) file /usr/sbin/httpd
Reading symbols from httpd...(no debugging
symbols found)...done.
(gdb) attach 386
Attaching to program '/usr/sbin/httpd', Pid 386
Reading symbols from /lib/libm.so.6...done.
Reading symbols from /lib/libcrypt.so.1...done.
Reading symbols from /lib/libdb.so.2...done.
Reading symbols from /lib/libdl.so.2...done.
```

```
Reading symbols from /lib/libc.so.6...done.  
Reading symbols from /lib/ld-linux.so.2...done.  
...
```

正如你从本例所看到的那样，`attach` 需要有一个进程的 ID 作为其参数，然后继续从被附加的程序链接的各种库里加载符号。重申一遍，当你完成了对被附加进程的检查后，使用 `detach` 命令以允许该进程继续执行，或者使用 `quit` 命令分离并退出 GDB。

提示： 如果你已经在 GDB 中运行了一个程序，而又要附加到另外一个运行的进程上，那么必须先杀死目前运行的这个程序。

8.4 小 结

本章只浮光掠影地介绍了 GDB 的一点表面功能。完全地介绍 GDB 功能需要几百页。正如本章一开始所说的，你在学习 GDB 上花的功夫越多，你从调试会话里获得的益处越大。在这一章里，你已经学会了怎样启动 GDB，已经看到了最简单地使用 GDB 所必须的基本命令，而且已经使用了 GDB 的一些高级特性。GDB 是你的朋友——学会使用它。

第9章 出错处理

无论算法速度多快或是代码编写得多好，你的程序最终还必须要有能够处理意外出错的功能。本章的目的是让读者熟悉出错处理工具并了解它们的用法。

9.1 出错处理简述

强健的、灵活的出错处理是正式发布的软件产品必不可少的功能。对出错条件考虑得越详细、处理得越周到，程序就越可靠。因此，在编程过程中，最好对每一个由库函数设置或返回的错误值都要进行检测并正确地处理它们。同样，自己编写的函数也应该返回有意义的、可以检测的错误代码。对错误的处理应该是尽可能消除错误并让程序继续运行。如果错误已经导致程序不能继续运行，在停止程序的运行之前应当向用户提示诊断信息或者将诊断信息写入日志文件中。最后，在不得不中止程序的运行之前，尽可能完成以下一些操作：关闭所有被打开的文件、更新永久性的数据（如配置文件）、尽量按顺序给用户显示结束的过程。最糟糕的情况就是一个程序无端死掉没有任何警告和出错提示信息。

9.2 出错处理选项

任何代码段对出错处理都可以有几种选择。第一种选择是不做处理，这在编程上等同于对困难的漠视和妥协。这是最不可取的一种办法，因为不对检测到的错误做任何处理、不向用户显示任何有用的信息，可能会造成不良的副作用，比如数据丢失或者让一个终端会话处于不可用状态。实际上，什么也不做的程序甚至不去检测可能的错误，比如无法打开文件或分配内存等。

第二种选择是，如果可能则检测错误并且向用户提供有关信息。C 语言有几种特性支持这种选择——这些特性将在下一节讨论。函数 `assert` 只能检测错误并终止程序执行。标准 C 库中的几个函数能显示与出现的错误有关的有用信息，而且还提供了能够让你控制程序何时和怎样终止的函数。

最好的一种出错处理选择是由 Linux 编程环境通过系统日志记录错误信息来提供的。这是一种极好的选择，因为它能让你永久地记录错误信息，这不但可以使得碰到程序错误时的调试工作更容易，而且可以使得碰到系统配置导致问题时的纠错工作更容易。

9.3 C 语言机制

ANSI C（如今也称为 ISO9899 C）标准中的几个特性支持出错处理。下面的章节将介绍 `assert` 例程，它通常由一个宏来实现，但被设计成像函数一样进行调用，同时还介绍几个可以用来编写自己的 `assert` 风格的函数的宏以及一些为检测和处理出错而设计的标准库函数。

9.3.1 `assert` 宏

宏 `assert` 的原型定义在头文件 `<assert.h>` 中，其作用是如果它测试的条件返回错误（即测试等于 0），则终止程序执行。原型定义如下：

```
#include<assert.h>
void assert(int expression) ;
```

`assert` 的作用是先计算表达式 `expression`，如果其值为假（即为 0），那么它首先向 `stderr` 打印一条出错信息，然后通过调用函数 `abort` 来终止程序运行。`expression` 可以是任何值为整数的有效 C 语句，比如 `fputs("some string",somefile)`。所以程序清单 9.1 中的示例程序将会在第二次调用 `assert` 时突然中断，因为第二次调用 `fopen` 函数时将执行失败（当然，除非你碰巧在当前目录下有一个名为 `bar_baz` 的文件）。

程序清单 9.1 使用 `assert`

```
/*
 * badptr.c - Testing assert
 */
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

int main(void)
{
    FILE *fp;

    fp = fopen("foo_bar","w"); /* This should work */
    assert(fp);
    fclose(fp);

    fp = fopen("bar_baz","r"); /* This should fail */
    assert(fp);
    fclose(fp); /* Should never get here */

    exit(EXIT_SUCCESS);
}
```

使用下面的命令行编译这个程序：

```
$ gcc badptr.c -o badptr
```

注意： 从本章开始，每章都提供一个用于其中示例程序的 makefile 文件。这个 makefile 文件位于对应章节的目录下。在大多数情况下，你只需简单地键入 make program 就能编译该程序，而不必输入一长串 gcc 命令行。例如，要构建 badptr，键入 make badptr 即可。

运行 badptr 的结果可能和下面类似：

```
$ ./badptr
badptr: badptr.c:17: main: Assertion 'fp' failed.
Aborted (core dumped)
```

这一输出显示：错误出现的程序是 badptr；assertion 失败的源代码文件为 badptr.c；调用 assert 失败的语句行号是 17（注意这不是真正出错的地方而是检测到出错的地方）；错误出现的函数以及出错的 assert 语句。如果你配置的系统具有转储内存（dump core）的功能，那么执行该程序的目录里将会出现一个 core 文件。

提示： 为了让系统上的程序能够转储内存，使用命令 ulimit -c unlimited。要了解有关 ulimit 命令的更多信息，可在命令行键入 help ulimit。

使用 assert 的缺点是，频繁调用它会极大地影响程序执行的速度。但是，偶尔的调用可能还是可以接受的。因为对性能有影响，所以很多程序员只是在开发阶段通过在 <assert.h> 的包含语句前插入 #define NDEBUG 来禁用 assert 调用，如下面的代码片段所示：

```
#include <stdio.h>
#define NDEBUG
#include <assert.h>
```

如果定义了 NDEBUG，则不会调用 assert 宏。

把 NDEBUG 定义为任何值（包括等于 0）都没有关系，只要有对它定义就能禁止对 assert() 的调用。使用这种方法必须注意一点，那就是 NDEBUG 在禁止了对 assert 调用的同时，也使得在 assert 中的表达式中隐含的某些操作不能被完成，而这些操作对后面的代码起着关键的作用。例如把函数调用放在 assert 的表达式中就会产生这种副作用（不要在 assert 中使用表达式），分析下面的代码：

```
assert ( (p = malloc( sizeof(char) * 100) == NULL) );
```

只要定义了 NDEBUG，assert 就不会被调用，p 永远不可能被初始化，在以后的代码中使用变量 p 就会出错。因此，使用 assert() 的正确的方法应该像下面这段代码那样：

```
p = malloc( sizeof(char) * 100);
assert(p);
```

即使是 assert 被禁止了，分配内存的操作还是会照样进行，当然对变量 p 还需要加上另外的检测。但是如果在 assert 语句中嵌入了 malloc 调用语句，则它永远不会被调用。

正如读者所见，assert 是很有用的，但它只提供了一种粗糙的终止程序运行的方式，这不是理想的方式。理想的方式应当是“适当的降级”，只有在不同层次的出错处理都失败了别无选择的情况下才终止程序的运行。在不得不警告或提示用户之前能成功处理的出错越

多，面对你的用户，程序也就越健壮。

9.3.2 使用预编译

除了 `assert` 函数外，C 标准还定义了两个宏：`__LINE__` 和 `__FILE__`，它们在许多程序执行时出错的场合下都很有用。例如，可以把它们和 `assert` 联用来更精确地定位导致 `assert` 失败的出错点。实际上，大多数 `assert` 实现都使用了 `__LINE__` 和 `__FILE__` 来完成它们的工作。程序清单 9.2~9.4 声明、定义并演示了一个用于打开文件的更强壮的函数 `open_file`，它就使用了 `__LINE__` 和 `__FILE__` 宏。

程序清单 9.2 `filefcn.h` 声明了函数 `open_file`

```
/*
 * filefcn.h - A function to open files
 */
#ifndef FILEFCN_H_
#define FILEFCN_H_

int open_file(FILE **fp, char *fname, char *mode, int line, char
*file);

#endif /* FILEFCN_H_ */
```

这里没有什么特别的地方。头文件 `filefcn.h` 声明了一个将在 `filefcn.c` 中进行定义的函数。为了避免因头文件在同一项目中多个编译单元中出现编译错误或警告，用 `#ifdef` 语句将 `open_file` 函数的声明包含起来，以确保预处理器只能遇到一次声明。

程序清单 9.3 `filefcn.c` 定义了函数 `open_file`

```
/*
 * filefcn.c - Using __LINE__ and __FILE__
 */
#include <stdio.h>
#include "filefcn.h"

int open_file(FILE **fp, char *fname, char *mode, int line, char *file)
{
    if ((*fp = fopen(fname, mode)) == NULL) {
        fprintf(stderr, "[%s:%d] open_file() failed\n", file, line);
        return 1;
    }
    return 0;
}
```

这文件定义了 `open_file` 函数。这里依然没有特别的地方，但是注意参数 `line` 和 `file` 分别是宏 `__LINE__` 和 `__FILE__` 的占位符，这两个宏将由调用函数传入。

程序清单 9.4 使用 `open_file` 的 `testmacs.c`

```
/*
 * testmacs.c - Exercise the function defined in filefcn.c
 */
#include <stdio.h>
#include <stdlib.h>
#include "filefcn.h"

int main(void)
{
    FILE *fp;

    /* This call will work */
    if(open_file(&fp, "foo_bar", "w", __LINE__, __FILE__)) {
        exit(EXIT_FAILURE);
    } else {
        fputs("This text proves we scribbled in the file.\n", fp);
        fclose(fp);
    }
    /* This call will fail */
    if(open_file(&fp, "bar_baz", "r", __LINE__, __FILE__)) {
        exit(EXIT_FAILURE);
    } else {
        fclose(fp);
    }

    return 0;
}
```

键入 `make testmacs` 或使用下面的命令来编译此程序：

```
$ gcc testmacs.c filefcn.c -o testmacs
```

在编译之前，预处理器把 `__LINE__` 分别换成 13 和 21，把 `__FILE__` 换成源代码文件的名称 `testmacs.c`。如果 `open_file` 函数调用成功，则它返回 0，否则它向 `stderr` 打印出诊断信息，指出它失败并返回 1 的文件名和行号(调用它的函数里的行号)。如果我们在 `open_file` 的定义中使用 `__LINE__` 和 `__FILE__`，行号和文件名就不太有用了。正如我们所定义的那样，你就能正确地了解哪里的函数调用失败了。

执行程序

```
$ ./testmacs
[testmacs.c:21] open_file failed
```

你可以看到它在 `testmacs.c` 的第 21 行失败，这正是我们预料的结果。`__FILE__` 和 `__LINE__` 在跟踪程序错误方面也很有用。学会使用它们。回忆第 3 章的内容，你也使用了 GNU C 的扩展 `__FUNCTION__` 变量来增加调试能力。在带有深层的函数嵌套调用的复杂程序中，使用 `__LINE__`、`__FILE__` 和 `__FUNCTION__` 这 3 个工具在跟踪模糊不清的错误时具有不可估量的作用。

9.3.3 标准库函数

在这里,“标准库”是指作为任何支持 ANSI/ISO C 标准的 C 环境一部分的变量、宏和函数。本节将深入介绍五个函数和一个变量(这听上去好像一部关于编程的电影“五个函数和一个变量”),它们在任何出错处理活动的重要部分。下面列出它们的原型和声明它们的头文件:

stdlib.h	void abort(void);
stdlib.h	void exit(int status);
stdlib.h	int atexit(void (*fcn)(void));
stdio.h	void perror(const char *s);
string.h	char *strerror(int errnum);
errno.h	int errno;

下面三个函数也是出错处理工具集的基本组成部分,它们的声明在头文件<stdio.h>中,其函数原型定义如下:

```
void clearerr(FILE *stream);
```

清除 EOF (end-of-file) 条件以及任何为 stream 设置的出错标志。

```
int feof(FILE *stream);
```

如果设置了 stream 的 EOF 标志则返回真 (非 0)。

```
int ferror(FILE *stream);
```

如果设置了 stream 的出错标志则返回真 (非 0)。

理解 errno

Linux 系统调用(要求内核服务的函数)和许多但不是所有的库函数(这类库函数大多数都在数学库中)在出错时都要把全局变量 `errno` 设置为一个非 0 值。但是,没有哪个库函数能把 `errno` 清零(设置 `errno=0`),而有时 -1 是一个合法的返回值,虽然通常情况下表示一种出错条件。所以为了避免出现虚假的出错情况,最好在调用一个可能设置 `errno` 变量的库函数之前首先把 `errno` 清零。程序列在程序清单 9.5 中。

程序清单 9.5 使用 `errno` 变量

```
/*
 * errno.c - Clearing errno between library calls
 */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <math.h>

int main(void)
{
    double d;
```

```
d = sqrt((double)-1);
if(errno) {
    perror("sqrt(-1) failed");
    errno = 0;
} else {
    printf("sqrt(-1) = %f\n", d);
}

d = sqrt((double)2);
if(errno) {
    perror("sqrt(2) failed");
    errno = 0;
} else {
    printf("sqrt(2) = %f\n", d);
}

exit(EXIT_SUCCESS);
}
```

注意： 本章的下一节讨论这个程序中使用的 `perror` 函数。

使用 `gcc errno.c -o errno -lm`（必须和数学库 `math` 链接，因为代码使用了 `sqrt` 函数），或者在命令行键入 `make errno` 来编译该程序。该程序的运行结果如下：

```
$ ./errno
sqrt(-1) failed: Numerical argument out of domain
sqrt(2) = 1.414214
```

为了演示必须把 `errno` 变量清零以避免虚报出错的情况，将第一个 `errno=0;` 语句注释掉或删除，重新编译后再运行该程序。这一次程序的输出是：

```
$ ./errno
sqrt(-1) failed: Numerical argument out of domain
sqrt(2) failed: Numerical argument out of domain
```

显然，2 处于 `sqrt` 函数的定义域，所以在调用可能设置 `errno` 变量的函数之前把 `errno` 清零的做法是必要的。

在上面例子中 `errno` 的值是 33，它代表 `EDOM` 宏。在数学库中的函数经常会把 `errno` 设置为 `EDOM` 和 `ERANGE`。当传递给函数的参数超出了函数的定义域时会产生 `EDOM` 错误。例如，`sqrt(-1)` 将产生域错误。当数学库中的函数返回值太大不能用双精度数表示时会出现 `ERANGE` 错误。`Log(0)` 将产生范围错误因为 `log(0)` 没有定义。有些函数既能产生 `EDOM` 错误也能产生 `ERANGE` 错误，所以可以把 `errno` 同 `EDOM` 和 `ERANGE` 比较以确定发生了哪一种错误以及怎样继续执行下去。`<errno.h>` 中还定义了许多其他错误，而且可以想像不只是数学库能够使用它们。表 9.1 列出了 126 种定义的错误中最常见的一些。

表 9.1 变量 `errno` 的常见值

宏	值	含义
<code>EPERM</code>	1	操作不被允许
<code>ENOENT</code>	2	文件或目录不存在
<code>ESRCH</code>	3	进程不存在
<code>EINTR</code>	4	系统调用中断
<code>EIO</code>	5	I/O 错误
<code>ENXIO</code>	6	设备或地址不存在
<code>E2BIG</code>	7	参数太长
<code>EBADF</code>	9	错误的文件号
<code>ECHILD</code>	10	子进程不存在
<code>EAGAIN</code>	11	重试
<code>ENOMEM</code>	12	没有内存
<code>EACCES</code>	13	没有权限
<code>EFAULT</code>	14	地址错误
<code>EBUSY</code>	16	设备或资源忙
<code>EEXIST</code>	17	文件存在
<code>ENODEV</code>	19	设备不存在
<code>ENOTDIR</code>	20	不是目录
<code>EISDIR</code>	21	是目录
<code>EINVAL</code>	22	无效参数
<code>EMFILE</code>	24	打开的文件太多
<code>ENOTTY</code>	25	不是打印机
<code>EFBIG</code>	27	文件太大
<code>ENOSPC</code>	28	磁盘上没有空间
<code>EPIPE</code>	32	管道中断
<code>EDOM</code>	33	数学参数超出函数定义域
<code>ERANGE</code>	34	数学结果不可表示
<code>EILSEQ</code>	84	非法字节序列
<code>ERESTART</code>	85	中断的系统调用应该重启
<code>EUSERS</code>	87	用户数太多

例如，当一个普通用户试图使用 `chmod` 命令给属于另一个用户的文件分配权限时会出现 `EPERM` 错误。当你试图杀死一个不存在的进程时会出现 `ESRCH` 错误。当一个程序不能分配更多的内存时会出现 `ENOMEM` 错误。当你试图拆卸 (`unmount`) 一个文件系统而另一个进程正在使用该文件系统中的某个文件时会出现 `EBUSY` 错误。程序员经常会碰到 `EINVAL` 错误，它意味着系统或库调用接收到一个类型或值无效的参数。`EMFILE` 意味着系统用完了 `inode` 节点，或是打开的文件数目达到了内核定义的 `FILE_MAX` 的值。`ENOSPC` 通常表明磁盘满了。有趣的是，所有这些错误中只有 `EDOM`、`ERANGE` 和 `EILSEQ` 是 ANSI/ISO C 标准定义的；而更多的错误是 POSIX 标准定义的。有的错误只针对多种 UNIX 版本和 Linux。

使用 abort 函数

abort 函数是一个比较严厉的函数。当调用它时，它会导致程序异常终止，所以程序在被终止前不能进行一些常规的清除工作，比如把终端窗口恢复到先前的状态并执行 atexit 登记的函数（atexit 在本章后面介绍）。abort 函数的原型已经在本节开头给出过。

但是，abort 会向操作系统返回一个预先定义的值以便告诉操作系统这是一个不成功的终止。如果没有 ulimit 的限制，abort 还会卸下一个 core 文件，以便事后进行调试。为了使调试更容易，我建议你在开发阶段总是用调试选项编译代码（使用 gcc -g 或者 -ggdb 选项），随后在发布代码前关闭调试选项重新编译程序或者用去除可执行文件。前面讨论过的 assert 调用了 abort 函数来终止调用它的程序。

注意： 用 strip 命令处理一个程序意味着用 strip 命令去除编译好的程序中的符号，这些符号通常是调试符号。这样做减少了程序对磁盘和内存的占用量，但不好的副作用是让调试变得更困难，许多软件商去除二进制代码是为了使其更困难，即使采用逆向工程方法回溯原来的程序也不是不可能。在开发源代码的社群里，能够访问源代码的现实自然让逆向工程的做法成了一种尚有争论的事。

程序清单 9.6 给出了 abort 函数的范例。

程序清单 9.6 使用 abort 函数

```
/*
 * boom.c - abort in action
 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    puts("About to abort...\n");
    abort();
    puts("I reckon it worked.\n"); /* Never get here */
    exit(EXIT_SUCCESS);
}
```

使用命令 gcc boom.c -o boom 或者调用 make boom 编译这个程序。在一个可以转储内存的系统里所做的一次运行结果如下：

```
$ ./boom
About to abort ...
Aborted (core dump)
```

如果你的系统不运行内存转储，则输出结果略有不同。但是，关键在于观察到第二个 puts 语句（以及 exit 语句）没有执行。程序 boom 突然退出了，表明它因一个未知原因而终止了。

使用 exit 函数

exit 是 abort 较文雅的兄弟。和 abort 一样, exit 也终止一个程序的执行并向操作系统返回一个值,但是和 abort 不同,它在完成清理工作后才终止程序,而且如果你还有其他由 atexit 登记的函数执行的清理任务,它也会调用它们。函数 exit 的原型是

```
#include <stdlib.h>
void exit(int status);
```

函数 exit 本身并不返回值,因此返回类型为 void。然而 status 是 exit 函数返回给操作系统的退出值。任何整数值都是合法的,但是在<stdlib.h>中只定义了 EXIT_SUCCESS 和 EXIT_FAILURE,而 0 是可移植的返回值。如果称不上大多数的话,本书中你所见到的许多程序都用到了它。例如,程序清单 9.6 的最后一行。

使用 atexit 函数

函数 atexit 登记在程序正常终止时要调用的函数,不是由 exit 调用就是通过程序的 main 函数返回来调用。atexit 的原型如下:

```
#include<stdlib.h>
int atexit(void (*function) (void));
```

传递给 atexit 的函数不带任何参数也没有任何返回值(也就是说,类型为 void)。你可以使用 atexit 来保证在程序正常关闭前执行某些代码。在讨论 abort 时已讲过,如果执行 abort,则不会调用由 atexit 登记的函数。如果 function 登记成功,则 atexit 返回 0,否则返回 1。程序清单 9.7 描述了 atexit 的用法。

程序清单 9.7 用 atexit 登记 exit 函数

```
/*
 * exitfcn.c - Fun with atexit
 */
#include <stdio.h>
#include <stdlib.h>

void f_atexit(void)
{
    puts("This message from f_atexit()");
}

int main(void)
{
    puts("This message from main()");

    if(atexit(f_atexit) != 0) {
        fprintf(stderr, "Failed to register f_atexit()\n");
        exit(EXIT_FAILURE);
    }

    puts("Exiting...");
```

```
    exit(EXIT_SUCCESS);  
}
```

if 语句是关键代码。我们通过提供一个几乎为空的函数 `f_atexit`，把 `f_atexit` 传递给 `atexit` 并检测返回值来判断函数登记是否成功。运行该程序以证实在 `main` 返回时调用了 `atexit`：

```
$ ./exitfcn  
This message from main()  
Exiting...  
This message from f_atexit()
```

正如你所看到的那样，在 `exit` 语句前的最后一条语句执行过后，接着就执行了 `f_atexit` 函数，它尽职地写下自己的输出。`atexit` 调用提供了一种在程序每次正常退出前保证执行某些代码的便捷途径。而且，可以使用 `atexit` 登记多个函数。但是必须记住，这些函数将以登记它们的逆序执行，就好像它们是从一个栈中弹出来的一样。

使用 `strerror` 函数

如果出现了错误，让你的用户（以及你本人）知道操作系统出了什么问题可能会比较有帮助。输入 `strerror`。正如你从它的原型定义中看到的那样

```
#include <string.h>  
char *strerror(int errnum);
```

`strerror` 返回一个指向字符串的指针，该字符串描述了和 `errnum` 相关的错误代码。所以，如果你把 `errno` 传递给 `strerror`，就可以得到可读的提示信息，而不再是一个冰冷而看不懂的数字。一会儿你就会看到一个使用 `strerror` 的例子。

使用 `perror` 函数

`perror` 调用能够方便地打印系统错误信息。它的原型是

```
#include <stdio.h>  
#include <errno.h>  
void perror(const char *s);
```

如果你的代码让系统调用失败，则系统调用返回-1 并且设置变量 `errno` 为一个说明上次错误的值，这和许多库函数是一样的。`perror` 首先打印字符串参数 `s`，后跟一个冒号和一个空格，然后是对应 `errno` 的错误信息和一个换行符。所以，调用

```
perror("Oops");
```

和调用

```
printf("Oops: %s\n",strerror(errno));
```

是等同的。程序清单 9.8 示范了 `strerror` 和 `perror` 的用法。用 `gcc errs.c -o errs -lm` 或 `make errs` 编译该程序。

程序清单 9.8 使用 `strerror` 和 `perror`

```
/*
 * errs.c - Using perror and strerror
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <errno.h>

int main(void)
{
    double d;
    char *p;

    errno = 0;
    d = sqrt(-1);
    if(errno) {
        p = strerror(errno);
        fprintf(stderr, "sqrt(-1): %s\n", p);
    }

    errno = 0; /* Reset errno to catch true errors */
    d = sqrt(-2);
    if(errno)
        perror("sqrt(-2)");

    exit(EXIT_SUCCESS);
}
```

当运行 `errs` 时，你不能区分出 `perror` 的输出和使用 `strerror` 输出的不同之处。你应该使用哪一种呢？这依赖于具体情况。如果你对 `perror` 短小而充分的输出感到满意，则使用 `perror`。另一方面，如果你希望构建定制的出错处理库，则使用 `strerror` 精心设计满足需要的错误信息。大多数开发人员都会满足于使用 `perror`。本书中的程序几乎专门使用 `perror`。

```
$ ./errs
sqrt(-1): Numerical argument out of domain
sqrt(-2): Numerical argument out of domain
```

9.4 使用系统日志

本章已经几次提到过写入日志消息。好的一面是你不必自己编写代码实现这个功能。Linux 用两个守护进程 `klogd` 和 `syslogd` 提供了集中的系统日志功能。我们将把注意力放在 `syslogd` 上，因为它控制着来自用户空间程序的消息的产生。`klogd` 供内核和运行在内核空间的程序，特别是设备驱动程序所使用（即使 `klogd` 可以使用 `syslogd`）。在学习了能够影响

如何以及何时写入系统日志消息的选项之后，你会看到系统日志（以后称为 syslog）的接口以及如何使用这个接口。

9.4.1 系统日志选项

在大多数 Linux 系统上，系统日志位于 `/var/log` 目录下。随你使用的 Linux 发布版本的不同，这些日志文件包括 `messages`、`debug`、`mail`、`news`，可能还有 `secure`。还可能其他的日志，这取决于在 `/etc/syslog.conf` 中定义的日志功能配置。标准的控制台日志守护进程是 `syslogd`，由它来维护这些日志文件。写入系统日志的消息由它的级别（level）和功能（facility）来控制，级别指出了消息的严重性或重要性，而功能告诉维护系统日志的 `syslogd` 守护进程是哪个程序发送的这条消息。一条日志消息的级别和功能合起来被称为它的优先级（priority）。正如你在本节介绍 syslog 接口的部分中所看到的那样，一条消息的优先级控制着它将出现在哪个日志文件中（取决于 `/etc/syslog.conf` 如何配置日志功能），甚至控制着它是否会出现。表 9.2 以降序列出了可能的日志级别；也就是说，从最严重的级别到最不严重的级别。表 9.3 列出了有效的系统日志功能名称。

表 9.2 syslog 的日志级别

级别	严重性
LOG_EMERG	系统不可用
LOG_ALERT	要求立即处理
LOG_CRIT	重大错误，比如硬盘故障
LOG_ERR	错误条件
LOG_WARNING	警告条件
LOG_NOTICE	正常但重要的消息
LOG_INFO	纯粹的通报消息
LOG_DEBUG	调试或跟踪输出

表 9.3 syslog 功能值

功能	消息源
LOG_AUTHPRIV	私有的安全和授权消息
LOG_CRON	时钟守护进程（ <code>crond</code> 和 <code>atd</code> ）
LOG_DAEMON	其他系统守护进程
LOG_KERN	内核消息
LOG_LOCAL[0-7]	为本地/站点使用而保留
LOG_LPR	打印机子系统
LOG_MAIL	邮件子系统
LOG_NEWS	新闻组子系统
LOG_SYSLOG	<code>syslogd</code> 产生的内部消息
LOG_USER	（默认值）一般用户级消息
LOG_UUCP	<code>uucp</code> 子系统

在大多数情况下，最好使用默认值，`LOG_USER` 级的功能。当然如果你正在编写一个邮件或新闻组客户端程序则例外。但是，如果你的站点的系统管理员设置了本地的功能级

别 LOG_LOCAL[0-7]，那么如果它们可用，则可以使用其中之一。选择正确的级别略有些技巧。通常应使用介于 LOG_ERR 和 LOG_INFO 之间的级别，但要认识到，如果你发送了一个 LOG_ERR 级别的消息，那么它会在所有用户终端和系统控制台上显示出来，而且还会向机器的系统管理员发送一页消息。希望的隐含规则应该是：选取与消息内容相适应的级别值。通常，当出现错误时，对于用户级程序来说 LOG_WARN 已经足够了，而 LOG_INFO 对于日常烦人的日志消息最合适。如果出现了可能致命的系统错误才使用 LOG_ERR(以上)的级别。但是没有硬性的和快速的规定，所以这些仅仅是建议。

9.4.2 系统日志函数

头文件 <syslog.h> 定义了到 syslogd 的接口。要创建一个日志消息，可使用 syslog 函数，其原型为

```
#include <syslog.h>
void syslog(int priority, char *format, ...);
```

priority 是表 9.2 和 9.3 分别列出的级别和功能的位逻辑“或”值。format 指定写入日志的消息和任何类似 printf 的格式说明字符串。特殊的格式说明字符 %m 由 strerror 为 errno 分配的错误消息（就好像调用了 strerror(errno) 函数一样）进行替换。

所以，综合起来，假如你在打开一个文件时发生了错误。则 syslog 调用类似于：

```
syslog(LOG_WARNING | LOG_USER, "unable to open file %s ***
%m\n", fname);
```

这里 fname 是尝试打开的文件名。这个调用在 /var/log/messages 中产生如下消息：

```
Mar 26 19:36:25 hoser syslog: unable to open file foo
*** No such file or directory
```

格式说明符 %m 附加上了字符串 “No such file or directory”，就好像调用了 strerror(errno) 一样。因为 LOG_USER 是默认的功能级，上面的代码片段也可以写成：

```
syslog(LOG_WARNING, "unable to open file %s *** %m\n", fname);
```

类似地，如果你只是希望草草写入一条不要紧的日志项，用

```
syslog(LOG_INFO, "this is a normal message \n");
```

它写入 /var/log/messages 的消息如下：

```
Mar 26 19:29:03 hoser syslog: this is a normal message
```

前面的例子有个问题，产生的日志消息不足以在能大到七、八兆字节的日志文件中惟一定位。能够用 openlog 来补救：

```
#include <syslog.h>
void openlog(const char *ident, int option, int facility);
```

facility 是表 9.2 中的某个值。ident 指定了加到日志消息前的字符串。option 是表 9.4 中零个或多个选项的位逻辑“或”值。

表 9.4 openlog 选项

选项	描述
LOG_PID	在每条消息中包含 PID（进程号）
LOG_CONS	如果消息不能写入日志文件，则发到控制台
LOG_NDELAY	立即打开连接（默认是在 syslog 第一次被调用时才打开连接）
LOG_PERROR	把消息写入日志文件的同时也输出到 stderr

注意：<syslog.h>也定义了 LOG_ODELAY，其含义是推迟建立与 syslog 的连接直到第一次调用 syslog。在 Linux 中，这个取值不产生任何作用，因为 LOG_ODELAY 是默认的动作。

函数 `openlog` 为 `syslog` 分配并打开一个（隐藏的）文件描述符。说该文件描述符是隐藏的是因为在 `syslog` 的公开接口中不能直接访问它。你只要相信它存在就行了。

`openlog` 的作用是定制日志操作。但是，`openlog` 是可选的——如果你自己不调用它，则 `syslog` 在你的程序第一次调用 `syslog` 函数时会自动调用它。与之相配对的函数 `closelog` 也是可选的，它只是关闭 `openlog` 打开的文件描述符。为了示范 `openlog` 的用法，考虑下面两行代码：

```
openlog("my_program", LOG_PID, LOG_USER);
syslog(LOG_NOTICE, "Pay attention to me!\n");
```

这段代码在 `/var/log/messages` 文件中产生如下信息：

```
Mar 26 20:11:58 hoser my_program[1354]: Pay attention to me!
```

而接下来的代码

```
openlog("your_program", LOG_PID, LOG_USER);
syslog(LOG_INFO, "No, ignore that other program!\n");
```

产生的输出为：

```
Mar 26 20:14:28 hoser your_program[1363]: No, ignore that other
program!
```

注意 `ident` 字符串和 `PID` 替换了功能（`facility`）字符串。这清楚地揭示了什么程序占有什么日志消息。事实上，`openlog` 为用户程序将来调用 `syslog` 而把默认的功能（`facility`）名设置为 `facility`。

类似地调用 `setlogmask` 为所有的日志消息设置了默认的级别：

```
#include <syslog.h>
int setlogmask(int priority);
```

在这里参数 `priority` 不是单个优先级就是优先级的范围。`setlogmask` 设置了优先级默认的掩码；`syslog` 拒绝任何没有在掩码中设置的优先级的消息。为了简化优先级掩码的设置工作，<syslog.h>中还定义了两个更有帮助的宏：

```
LOG_MASK(int priority)
```

和

```
LOG_UPTO(int priority)
```

LOG_MASK 创建一个仅由一个优先级组成的掩码, priority 作为参数传递给该宏。另一方面, LOG_UPTO 创建一个由一系列降序优先级组成的掩码, 这里 priority 是允许的最低优先级。例如, LOG_UPTO (LOG_NOTICE) 创建的掩码包含了从 LOG_EMERG 到 LOG_NOTICE 之间的任何级别的消息。而 LOG_INFO 或 LOG_DEBUG 级别的消息则不能通过。查看程序清单 9.9, 它示范了 syslog 接口的用法。

程序清单 9.9 使用 syslog 接口

```
/*
 * logit.c - Demonstrate the syslog interface
 */
#include <syslog.h>
#include <stdio.h>
#include <unistd.h> /* getuid() */
#include <sys/types.h> /* getuid() */
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int omask; /* The old priority mask */

    openlog("logit", LOG_PID, LOG_USER);
    syslog(LOG_INFO, "This message courtesy of UID %d\n", getuid());
    syslog(LOG_NOTICE, "Hopefully, you see this\n");
    closelog(); /* Reset the facility */

    /* Don't want to see DEBUG and INFO messages */
    omask = setlogmask(LOG_UPTO(LOG_NOTICE));

    /* Save the old mask */
    syslog(LOG_INFO, "You should not be seeing this\n");
    syslog(LOG_DEBUG, "I hope you don't see this\n");
    syslog(LOG_NOTICE, "Restoring the old priority mask\n");

    /* Restore original priority mask */
    setlogmask(omask);
    syslog(LOG_INFO, "You should see this\n");
    syslog(LOG_DEBUG, "I hope you see this\n");

    exit(EXIT_SUCCESS);
}
```

注意: 函数 getuid 返回调用过程的“有效”(effective)用户 ID (UID)。它将在第 13 章介绍。

代码的第一段用 LOG_PID 选项和标准的 LOG_USER 功能打开日志文件, 随便向日志文件中写入两条日志消息, 然后关闭它。代码第二段使用 setlogmask 函数控制优先级掩码,

把它设置为不能写入带有级别 LOG_INFO 和 LOG_DEBUG 的消息。注意 omask 保存了原来的优先级掩码，于是能在第三段代码中恢复。编译并执行该程序，logit 在 /var/log/messages 文件中产生了如下消息：

```
Jun 17 10:20:34 hoser logit[6037]: This message courtesy of UID #1000
Jun 17 10:20:34 hoser logit[6037]: Hopefully, you see this
Jun 17 10:20:34 hoser logit[6037]: Restoring the old priority mask
Jun 17 10:20:34 hoser logit[6037]: You should see this
```

成功了！在首次改变了优先级掩码后，带有 LOG_INFO 和 LOG_DEBUG 级别的消息不能通过，但是带有更高级别的消息，比如 LOG_NOTICE 能够很好地通过。在恢复了原来的优先级掩码后，LOG_INFO 消息如预期的那样也能写入日志。但是 LOG_DEBUG 又怎样呢？很简单。包括我的系统在内的许多系统配置 syslogd，把调试消息写入一个单独的文件中，在我的系统上这个文件是 /var/log/debug。快速查看一下该文件，其内容如下：

```
Jun 17 10:20:34 hoser logit[6037]: I hope you see this
```

但是，当 LOG_DEBUG 消息被掩蔽后发送的消息没有出现。

注意： 根据你的系统的配置不同，你可能会看到与此不同的输出结果。

在这一节中，你从 C 程序员的角度看到了 syslog 接口的功能。本章的最后一节将向你介绍一种让 shell 程序员同样具有使用系统日志服务能力的工具。

提示： 跟踪文件 /var/log/messages 的一种简单方法是打开一个单独的 xterm 窗口，使用 tail -f /var/log/messages 命令查看文件末尾的内容。每次有新消息写入日志文件时，它就会在屏幕上滚动显示出来。在大多数系统上，这样做需要有超级用户权限，因为只有超级用户有 /var/log/messages 文件的读写权。

9.4.3 用户程序

这里没有忘记 shell 程序员。Linux 通过用户级程序 logger 提供了系统日志的 shell 接口。logger 是面向 syslog 的 shell 和命令行接口。它的完整语法是

```
logger [ -s ] [ -f file ] [ -p pri ] [ -t tag ] [ -u socket ] [ message ...]
```

logger 的选项在表 9.5 中予以说明。

表 9.5 logger 命令的选项

选项	描述
-i	向日志消息中加入 PID
-s	同时向 stderr 和日志写入日志消息
-f file	向文件写入日志消息
-p pri	使用 pri 指定的优先级
-t tag	向日志消息加入字符串 tag
-u socket	向 socket 指定的套接口而不是系统日志写入日志消息
message	写入日志的消息

使用 -t 把调用 logger 的脚本名字写入日志消息。如果没有指定 message, 就会把标准输入当作日志记录下来。-p 的参数 pri 的形式是功能·级别 (facility.level) 对, 它们的值在表 9.2 和 9.3 中说明。默认值为 user.notice。程序清单 9.10 示范了如何在 shell 脚本中使用 logger。

程序清单 9.10 使用 logger 命令

```
#!/bin/sh
# logger.sh - Demonstrate the shell command interface to syslog
#####

echo 'Type the log message and press ENTER'
read MSG
logger -is -t logger.sh $MSG
```

如果你不理解脚本的语法, 不必担心, 它将在第 30 章中讨论。在提示要求输入一条日志消息之后, 脚本读入日志消息, 然后调用 logger 把它插入到日志文件中。-s 选项能让该消息写入日志文件的同时输出到标准出错文件上 (standard error, stderr), 而 -t 选项则让 logger.sh 将它的名字作为前缀加到每条消息的前面。该程序运行结果类似:

```
$ ./logger.sh
Type the log message and press ENTER
DEMONSTRATING THE LOGGER COMMAND
logger.sh[6160]: DEMONSTRATING THE LOGGER COMMAND
```

最后一行输出反映了 -s 选项的作用, 它把日志消息 (少了时间戳信息) 写到了标准出错文件上。而日志文件 /var/log/messages 中写入的内容为:

```
Jun 17 11:26:08 hoser logger.sh[6160]: DEMONSTRATING THE LOGGER
COMMAND
```

正如你所看到的那样, syslog 向日志文件写入了脚本的名字、PID 和消息。在有的系统上, 超过 80 个字符的日志消息 (包括时间戳和其他标识信息) 会被截短到 80 个字符。

9.5 小 结

阅读完本章, 读者应该对出错处理有了一个比较清楚的认识, 并且对管理错误状态的工具具有更深的理解。比较遗憾的是还没有提供出错处理的 API, 仅仅是在系统中分散地提供了一些工具。读者需要搜集这些工具并运用它们来编写健壮的容错软件。本章介绍了 C 语言提供的丰富的函数集, 如 asser、abort、exit、atexit 以及出错处理例程 perror 和 strerror。还介绍了系统日志工具并示范了它们的用法。

第 10 章 使 用 库

本章讲述创建和使用编程库，所谓的库就是可以被多个软件项目使用（和重用）的代码集。库是软件开发所追求的目标——代码重用的经典例子。它们把经常使用的编程例程集中到一起。系统 C 语言库就是一个例子。它包含了数百个经常使用的例程，比如输出函数 `printf` 和输入函数 `getchar`，如果你每次编写新程序时都要重写这些函数，一定会觉得很乏味。除了代码重用和编程人员使用方便两方面的优点外，库还提供了许多实用工具代码，例如用于网络编程的函数、图像处理函数、数据处理函数和系统调用。

10.1 使用编程库

如前所述，编程库有两个主要优点：它们能够实现代码重用，而且能够提供数百行经过测试和调试的工具代码。本节讨论有关库兼容性的一些问题，说明了“标准”库的命名和编号约定，而且列举并描述了许多在 Linux 编程中经常使用的库。

10.1.1 库兼容性

库兼容性是指在库的多个修订或升级版本间保持稳定一致的变量、数据结构、公共函数接口和总体功能。具有三年以上使用 Linux 经验的用户还会记得从成熟的 C 库 `libc5` 过渡到当前的 C 库 `libc6` 所遭受的重负和沮丧。

注意： 对于 Linux 新用户来说，从 `libc5` 过渡到 `libc6` 破坏了成百上千的应用软件，迫使许多软件重新编写并且导致更多的软件重新编译——为 `libc5` 编写的应用软件和库都不能在 `libc6` 的系统上运行，反之亦然。因为 C 库是 Linux 系统的基础，所以此次升级造成的影响是非常普遍的，而且有时又是极为严重和困难的。

当然，库和应用软件一样，随着不断补充新功能和不断调试而继续演化发展。关键在于要聪明地进行修改。然而，随着应用软件的成熟，接口仍旧保持基本稳定，而它们的特性既使有了扩充也应保持在本质上是相同的。这同样适用于库：必要时扩展和调试它们，但要在不破坏依赖于它们当前行为和接口的应用软件（或其他库）的前提下这么做。例如，如果你需要增加功能，不要改变现有的接口（函数），而是扩展现有接口（函数）的功能或者简单地提供所需的行为。这样做能够避免破坏依赖于当前接口的现有程序。

你怎样保持库的兼容性？显然，针对每个库都有不同的方法。但是，一般说来，以下情况会导致库版本不兼容：

- 导出函数接口变化。
- 增加了新的函数接口。

- 函数功能与最初的规定相比有了变化。
- 导出的数据结构变化。
- 增加了导出的数据结构。

相应地，采取以下方针保持库的兼容性：

- 给增加到库中的函数使用新名字，而不是改变现有函数或者改变它们的行为。
- 只向现有数据结构的末尾增加数据项，而且让新增加的项不是可选的就是在库内部初始化。
- 不要扩大数组中使用的数据结构。

从这里的讨论得出的另一个结论是缜密的软件设计（这对于需要重复使用的库来说尤其重要）要求在创建一个通用的、可扩展的公共接口上下功夫，这样做能够降低最终的修改会在修订版本间引入本质上不兼容的可能性。有时这种程度的改变不可避免，但目标是尽可能减少其导致的痛楚。

10.1.2 命名和编号约定

过去的若干年中，产生了对库进行命名和编号的小小的约定集合。命名约定相当简单。首先，正如在第3章提到的那样，所有的库名都以 `lib` 开头，这显然表示它们都是库。许多开发工具都依赖这个约定，特别是 `GCC`，它会在 `-l` 选项所指定的文件名前自动地插入 `lib`。第二个约定是文件名以 `.a`（代表存档，`archive`）结尾的库都是静态库（参见本章后面 10.3 节的内容）。文件名以 `.so`（大概代表共享目标文件，`shared object`）结尾的库都是共享库（在本章后面的 10.4 节中讨论）。比如，`libdl.a` 是一个静态库而 `libc.so` 是一个共享库。

编号约定略微复杂一些。一般格式为 `library_name.major_num.minor_num.patch_num`。例如，在笔者的系统上，`GNU` 数据库的全名为 `libgdbm.so.2.0.0`。把它分开看：

- `library_name` 是 `libc.so`
- `major_num` 是 2
- `minor_num` 是 0
- `patch_num` 是 0

通常约定，当库的变化达到了和以前的版本不能兼容的程度时就要增加主版本号（`major_num`）。当库有了新变化而又能和以前的版本保持兼容时就只改变次版本号（`minor_num`）。为修正库中的错误而进行改动则会改变补丁级别号（`patch_num`）。补丁级别号有时又称为发行号（`release number`）。所以，处于某个主修订版本之下的数据库名可以是 `libgdbm.so.2.0.0`，而错误修订发行版本可能会把次版本号增加到 1，所以它的名字就变成了 `libgdbm.so.2.1.0`。

下面介绍库命名的最后一个约定。当你浏览自己的系统时，你会碰到以 `_g` 和 `_p` 结尾的库，如 `libform_g.a` 和 `libform_p.a`。它们是本库的特殊版本——此时的基本库为 `libform.a`。通常名字以 `_g` 结尾的库是调试库，它们编入了特殊的符号和功能，能够增加对采用了这个库的应用程序进行调试的能力。类似地，代码剖析（`profiling`）库通常在名字后边附加 `_p`，而且它们包含的代码和符号能够进行复杂的代码剖析和性能分析。如果你使用了这些库中

的某一个，那么一旦完成了调试或剖析工作，需要使用正常库重新编译你的程序。

10.1.3 经常使用的库

表 10.1 列出了常用库、声明它们的公共接口的头文件以及对它们提供功能的简要描述。但是，这个列表并没有列举完所有的库。它只覆盖了主流 Linux 发布版本典型安装的那些库——快速搜索一下 Freshmeat 站点（<http://www.freshmeat.net/>）或者 Metalab（<ftp://metalab.unc.edu/pub/Linux/>）就能发现，实际上在因特网上散布着数以百计的库。

表 10.1 常用的 Linux 编程库

库	头文件	描述
libGL.so	<GL/gl.h>	实现到 OpenGL API 的接口
libGLU.so	<GL/glu.h>	实现到 OpenGL API 的接口
libImlib.so	<Imlib.h>	实现一套图像处理例程，许多人认为它比作为 X Window 系统一部分的标准 XPM (X pixmap) 库要好
libc.so		实现标准 C 库（不需要头文件）
libcom_err.so	<com_err.h>	常用出错处理例程的集合
libcrypt.so	<crypt.h>	加密函数的集合
libcurses.so	<curses.h>	光标字符模式的屏幕操作库（典型地，是到 libncurses.so 的符号链接）
libdb.so	<db.h>	创建和操作数据库的库
libdl.so	<dlfcn.h>	让程序在运行时加载和使用库代码而无须在编译时链接库
libform.so	<form.h>	实现字符模式应用程序的窗体处理能力
libgdbm.so	<gdbm.h>	GNU 数据库管理器，对 libdb.so 提供接口的改进版本
libglib.so	<glib.h>	Glib 库，提供了大多数程序需要的大量基本工具函数，比如散列表和字符串操作例程
libgthread.so	<glib.h>	实现对 Glib 的线程支持
libgtk.so	<gtk/gtk.h>	GIMP (GNU Image Manipulation Program) 下的 X 库，GIMP Tool Kit 的基础库
libhistory.so	<readline/readline.h>	实现 GNU readline (libreadline) 包中的命令行历史机制
libjpeg.so	<jpeglib.h>	定义到 JPEG 库的接口，赋予应用程序使用 JPEG 图像文件的能力
libm.so	<math.h>	实现标准 C 数学库
libmenu.so	<menu.h>	提供在字符模式屏幕上创建和使用菜单的例程
libncurses.so	<ncurses.h>	使用 ncurses 文本模式屏幕控制系统的应用程序的基础库
libnss.so	<nss.h>	用于名字服务切换工具的功能，提供了名字数据库比如 DNS 的接口
libpanel.so	<panel.h>	提供在字符模式屏幕上创建和使用面板的例程
libpbm.so	<pbm.h>	可移植的位图 (bitmap) 库，实现了使用多种格式单色位图的接口

(续表)

库	头文件	描述
libpgm.so	<pgm.h>	可移植的灰度图 (graymap) 库, 实现了使用多种格式灰色位图的接口
libpng.so	<png.h>	用于编码、解码和操作 PNG (Portable Network Graphics, 可移植的网络图形) 格式图像文件的参考实现
libpnm.so	<pnm.h>	可移植 anymap 库是使用多种位图格式, 如 PBM (monochrome bitmap, 单色位图)、PPM (color pixmap, 彩色像素图) 和 PGM (grayscale pixmap, 灰度像素图) 的基础库
libppm.so	<ppm.h>	可移植的像素图库实现了使用多种格式彩色像素图的接口
libpthread.so	<pthread.h>	实现 POSIX 线程库, 标准的 Linux 多线程库
libreadline.so	<readline/readline.h>	GNU readline 包的基础库, 该软件包能够让应用软件存储、记忆并且编辑命令行。bash shell 命令行的编辑特性就是其示例
libresolv.so	<resolv.h>	提供使用因特网域名服务器和服务的接口
libslang.so	<slang.h>	提供方便的脚本语言 S-lang, 用于嵌入其他应用程序
libtiff.so	<tiffio.h>	读写 TIFF 格式图像文件的库
libvga.so	<vga.h>	Linux 的底层 VGA 和 SVGA 图形库
libz.so	<zlib.h>	通用压缩例程库

10.2 库操作工具

在进入到创建和使用库的话题之前, 本节简单地浏览一下将要用来创建、维护和管理编程库的一些工具。若想了解接下来讨论的每个命令和程序的详细信息, 请查阅手册页面和信息文件。特别地, 你将要学习 nm、ar、ldd 和 ldconfig 的使用。

10.2.1 理解 nm 命令

命令 nm 列出编入目标文件或二进制文件的所有符号。一种用途是查看程序调用什么函数。另一个用途是查看一个给定的库或者目标文件是否提供了所需的函数。nm 使用下面的语法:

```
nm [options] file
```

nm 列出保存在 file 中的符号。表 10.2 描述了 nm 的一些有用的选项。

表 10.2 nm 的选项

选项	描述
-c --demangle	将符号名转换为用户级的名字。在让 C++ 函数名可读方面特别有用
-s --print-arnmap	当用于存档 (.a) 文件时, 输出把符号名映射到定义该符号的模块或成员名的索引

(续表)

选项	描述
<code>-u</code> <code>--undefined-only</code>	只显示未定义的符号——在被检查的文件外部定义的符号
<code>-l</code> <code>--line-numbers</code>	使用调试信息输出定义每个符号的行号，或者未定义符号的重定位项

10.2.2 理解 ar 命令

`ar` 命令用来操作高度结构化的存档 (archive) 文件 (包含其他文件，通常是目标文件)。该命令最常用来创建静态库——包含一个或多个目标文件，预编译格式的例程的目标文件称为成员 (member)。`ar` 也能创建和维护符号名的交叉索引表，如函数和变量名到定义它们的成员之间的交叉索引表。`ar` 的语法为

```
ar {dmpqrtx} [member] archive files ...
```

表 10.3 描述了最常用的 `ar` 选项。

表 10.3 `ar` 的选项

选项	描述
<code>-c</code>	如果存档文件不存在，则从多个文件创建存档文件，并且不显示 <code>ar</code> 发出的警告
<code>-s</code>	创建或升级从符号到定义它们的成员之间的交叉索引映射表
<code>-r</code>	向存档文件插入 <code>files</code> ，替换已有的任何同名成员。新成员添加到存档文件的末尾
<code>-q</code>	把 <code>files</code> 添加到存档文件末尾而不检查是否进行替换

提示： 对于用 `ar` 命令创建的存档文件，可以通过使用索引来加快访问它的速度。工具 `ranlib` 能够精确地完成此项工作，它把存档文件的索引保存在存档文件本身里。`ranlib` 的语法为

```
ranlib [-v|-V] file
```

这样做能在 `file` 中生成一个符号映射。这个命令和 `ar -s file` 命令等价。

在 10.3 节中将示范 `ar` 命令的用法。

10.2.3 理解 ldd 命令

`nm` 命令列出目标文件定义的符号，但如果你不知道哪个库定义哪个函数，`ldd` 命令就显得更有用。`ldd` 列出了为使程序正常运行所需要的共享库。其语法为：

```
ldd [options] file
```

`ldd` 输出 `file` 所要求的共享库的名字。例如，在笔者的系统上，邮件客户端程序 `Mutt` 需要的 5 个共享库如下：

```
$ ldd /usr/bin/mutt
libnsl.so.1 => /lib/libnsl.so.1 (0x40019000)
libslang.so.1 => /usr/lib/libslang.so.1 (0x4002e000)
libm.so.6 => /lib/libm.so.6 (0x40072000)
libc.so.6 => /lib/libc.so.6 (0x4008f000)
```

```
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

在你的系统上，所需库的列表可能略有不同。箭头左边一列显示了 Mutt 库所需的 so 文件名；箭头右边一列显示了库的真实名称。需要牢记的是，应用程序链接到库的 so 名字是到实际库的符号链接。

表 10.4 描述了一些 ldd 的有用选项。

表 10.4 ldd 的选项

选项	描述
-d	执行重定位并报告所有丢失的函数
-r	执行对函数和数据对象的重定位并报告丢失的任何函数或数据对象

10.2.4 理解 ldconfig

命令 ldconfig 使用以下语法：

```
ldconfig [options] [libs]
```

命令 ldconfig 决定位于目录/usr/lib 和/lib 下的共享库所需的运行的链接，这些链接在命令行上的 libs 指定并被保存在/etc/ld.so.conf 中。命令 ldconfig 和动态链接 / 装载工具 ld.so 协同工作，一起来创建和维护对最新版本共享库的链接。表 10.5 是 ldconfig 使用的一般选项，不带选项的 ldconfig 命令仅更新高速缓冲文件。

表 10.5 ldconfig 选项

选项	描述
-p	仅打印出文件/etc/ld.so.cache 的内容，此文件是 ld.so 所知道的共享库的当前列表
-v	更新/etc/ld.so.cache 的内容，列出每个库的版本号，扫描的目录和所有创建和更新的链接

10.2.5 环境变量和配置文件

动态链接器 / 加载器 ld.so 使用两个环境变量。第一个是\$LD_LIBRARY_PATH，一个由冒号分隔的目录清单，在这些目录下可以搜索运行时的共享库。你可以使用这个变量告诉 ld.so 在哪儿找到没有保存在标准位置的库，比如在你的主目录下保存的特殊库，它和环境变量\$PATH 很相似。第二个变量是\$LD_PRELOAD，一个由空格分隔的、附加的、用户指定的共享库，它需要在其他所有库加载之前加载。这样可以在其他共享库中有选择性地重载函数。

ld.so 还使用两个配置文件，这两个文件的目的和前面讲到的环境变量是平行的。除了标准目录/usr/lib 和/lib 以外，清单/etc/ld.so.conf 中列出了链接器 / 加载器搜索共享库时要查看的目录。/etc/ld.so.preload 是环境变量\$LD_PRELOAD 的基于磁盘的版本：它包含了一个在执行程序之前要加载的由空格分隔的共享库列表。

10.3 编写并使用静态库

静态库（说实在的，还有共享库）是包含了目标文件的文件，这些目标文件被称为模块或成员，是可以重用的预编译代码。它们以特殊的格式和一个表或者映射保存在一起，这个表或者映射将符号名和保存该符号的成员名链接起来。映射加速了编译和链接过程。静态库一般以扩展名.a（代表存档文件，archive）命名。

为了使用库代码，必须在源代码文件中包含适当的头文件并且链接到库。例如，程序清单 10.1（它是用于一个简单出错处理库的头文件）以及程序清单 10.2（它是其相应的源代码）。

注意： Richard Stevens 所著《Advanced Programming in the UNIX Environment》一书的读者会认出这段代码。我已经用它很多年了，因为它干净利落地满足了对基本出错处理例程的要求。在此我对 Stevens 先生慷慨地允许我复制这些代码表示感谢。

程序清单 10.1 一个简单的出错处理库的接口

```
/*
 * liberr.h
 * Declarations for simple error-handling library
 */
#ifndef _LIBERR_H
#define _LIBERR_H

#include <stdarg.h>

#define MAXLINELEN 4096

/*
 * Print an error message to stderr and return to caller
 */
void err_ret(const char *fmt, ...);

/*
 * Print an error message to stderr and exit
 */
void err_quit(const char *fmt, ...);

/*
 * Print an error message to logfile and return to caller
 */
void log_ret(char *logfile, const char *fmt, ...);

/*
 * Print an error message to logfile and exit
 */
void log_quit(char *logfile, const char *fmt, ...);
```

```

/*
 * Print an error message and return to caller
 */
void err_prn(const char *fmt, va_list ap, char *logfile);

#endif _LIBERR_H

```

程序清单 10.2 一个简单的出错处理库的实现

```

/*
 * liberr.c
 * Implementation of error-handling library
 */
#include <errno.h> /* errno */
#include <stdarg.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "liberr.h"

void err_ret(const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    err_prn(fmt, ap, NULL);
    va_end(ap);
    return;
}

void err_quit(const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    err_prn(fmt, ap, NULL);
    va_end(ap);
    exit(EXIT_FAILURE);
}

void log_ret(char *logfile, const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    err_prn(fmt, ap, logfile);
    va_end(ap);
    return;
}

void log_quit(char *logfile, const char *fmt, ...)
{

```

```

    va_list    ap;

    va_start(ap, fmt);
    err_prn(fmt, ap, logfile);
    va_end(ap);
    exit(EXIT_FAILURE);
}

extern void err_prn(const char *fmt, va_list ap, char *logfile)
{
    int      save_err;
    char buf[MAXLINELEN];
    FILE *plf;

    save_err = errno; /* value caller might want printed */
    vsprintf(buf, fmt, ap);
    sprintf(buf + strlen(buf), ": %s", strerror(save_err));
    strcat(buf, "\n");
    fflush(stdout); /* in case stdout and stderr are the same */
    if(logfile != NULL)
        if((plf = fopen(logfile, "a")) != NULL) {
            fputs(buf, plf);
            fclose(plf);
        } else
            fputs("failed to open log file\n", stderr);
    else
        fputs(buf, stderr);
    fflush(NULL); /* flush everything */
    return;
}

```

对这些代码作一简要说明可能会有帮助。liberr.h 和 liberr.c 包含了<stdarg.h>，因为它们使用了ANSI C的可变长度参数列表工具（如果你对可变长度参数列表不熟悉，请参看C参考手册）。为了防止多次包含头文件，我们把它包含到预处理宏_LIBERR_H中。不要在最终投入使用的代码中使用出错日志函数log_ret和log_quit；在<syslog.h>中定义的系统日志工具更适用一些。最后，不应该在一个作为守护进程运行的程序中使用这个库，因为它向stderr有输出。这种输出对于守护进程来说没有意义，因为守护进程通常没有控制终端。

建立静态库的第一步是把代码编译为目标文件形式：

```
$ gcc -c liberr.c -o liberr.o
```

接下来使用工具ar创建一个存档文件：

```
$ ar rcs liberr.a liberr.o
```

如果以上两步正常通过，静态库liberr.a就产生了。下一步是把一个程序和liberr.a链接起来。程序清单10.3给出了一个驱动程序errtest.c来测试产生的新库。

程序清单 10.3 出错处理库的驱动程序

```

/*
 * errtest.c - Test program for error-handling library
 */

#include <stdio.h>
#include <stdlib.h>
#include "liberr.h"

#define ERR_QUIT_SKIP 1
#define LOG_QUIT_SKIP 1

int main(void)
{
    FILE *pf;

    puts("Testing err_ret");
    if((pf = fopen("foo", "r")) == NULL) {
        err_ret("%s %s", "err_ret:", "failed to open foo");
    }

    puts("Testing log_ret");
    if((pf = fopen("foo", "r")) == NULL) {
        log_ret("errtest.log", "%s %s", "log_ret", "failed to open
            foo");
    }

    #ifndef ERR_QUIT_SKIP
    puts("Testing err_quit");
    if((pf = fopen("foo", "r")) == NULL) {
        err_quit("%s %s", "err_quit:", "failed to open foo");
    }
    #endif /* ERR_QUIT_SKIP */

    #ifndef LOG_QUIT_SKIP
    puts("Testing log_quit");
    if((pf = fopen("foo", "r")) == NULL) {
        log_ret("errtest.log", "%s %s", "log_quit:", "failed to open
            foo");
    }
    #endif /* LOG_QUIT_SKIP */

    exit( EXIT_SUCCESS);
}

```

测试程序很简单：它试着打开一个不存在的文件 4 次，每次采用一种库的出错处理函数。宏 `ERR_QUIT_SKIP` 和 `LOG_QUIT_SKIP` 避免 `_quit` 函数的执行。要测试它们，可以注释掉其中一个宏，重新编译并运行该程序。对另一个宏重复上述过程。

编译 `errtest.c` 的正确方法是用 GCC 的 `-static` 选项把它和 `liberr.a` 静态链接起来，做法如下：

```
$ gcc errtest.c -o errtest -static -L. -lerr
```

如果你没有指定-static 选项，那么 GCC 将使用动态链接创建 errtest，而这并不是我们此时希望的结果。-lerr 告诉 GCC 在当前命令下查找库文件 liberr。为了证实你已经静态链接了该程序，可以使用 file 命令：

```
$ file errtest
errtest: ELF 32-bit LSB executable, Intel 80386, version 1,
statically linked, not stripped
```

输出的关键字眼是“statically linked”（静态链接）。你也可以使用 nm 命令来显示二进制文件中的符号：

```
$ nm errtest
0804e1c0 W _Exit
0807add4 ? _GLOBAL_OFFSET_TABLE_
08079c00 D _IO_2_1_stderr_
08079a80 D _IO_2_1_stdin_
08079b40 D _IO_2_1_stdout_
0804a484 T _IO_adjust_column
0804a58c T _IO_cleanup
0804a258 T _IO_default_doallocate
0804a35c T _IO_default_finish
0804a82c T _IO_default_imbue
0804a6e4 T _IO_default_pbackfail
...
```

为了节省空间，输出内容被截断了。怎样中断输出呢？nm 默认的输出格式是一个由符号值、符号类型和符号名三列组成的列表。符号类型 T 的意思是该符号——例如，__IO_default_finish——出现在目标文件的文本或代码区域中。符号类型 U 的意思是此成员中未定义该符号。要注意的关键一点是 errtest 没有未定义的符号。你可以一页页地翻看 nm errtest 的完整输出，但不会看到任何未定义的符号，这意味着所有符号都在程序中定义，从而说明它是静态链接的。参看 binutils 的信息页面（info binutils nm）了解 nm 输出的完整说明。

errtest 的输出如下：

```
$ ./errtest
Testing err_reterr_ret: failed to open foo: No such file or directory
Testing log_ret
```

函数 log_ret 把它的输出写入 errtest.log 文件。对 errtest.log 使用 cat 命令得到如下结果：

```
$ cat errtest.log
log_ret: failed to open foo: No such file or directory
```

对几个*_quit 函数的测试留给读者作为练习。

10.4 编写并使用共享库

共享库和静态库相比有几个优点。第一，共享库占用的系统资源更少。由于共享库并没有被编译进每个二进制文件中，只是在运行时从单个文件——共享库链接加载，所以它们占用更少的磁盘空间。因为内核把该库所占用的内存在使用该库的所有程序间共享，所以它们使用的系统内存更少。第二，共享库最低限度也比静态库快许多，因为它们只需要向内存里加载一次。第三，共享库使得代码维护的工作大大简化。当修正了一些错误或者添加了一些特性时，用户只要获得升级后的库并安装它即可。而使用静态链接库的程序每次修改都要重新编译连接到新库。

共享库是怎样工作的呢？正如前面所介绍的那样，在运行时动态链接器/加载器 `ld.so` 把二进制文件中的符号名链接到适当的共享库上。共享库有一个特殊的名字，即 `so` 名字，它由库名和主版本号组成。应用程序链接到 `so` 名字。`ldconfig` 工具创建一个从实际库到 `so` 名字的符号链接。例如，老的 C 库的 `so` 名字为 `libc5`，在笔者的系统上是 `libc.so.5`。而实际的名字是 `libc.so.5.4.46`。`ldconfig` 创建了从库文件到 `so` 名字的符号链接，并且把这些信息保存在缓冲文件 `/etc/ld.so.cache` 中。在运行时，`ld.so` 读取该缓冲，找到所需的 `so` 名字，把实际的库（因为采用了符号链接）加载到内存中，并且把应用程序中的函数调用链接到加载库中合适的符号上。

创建共享库的方法和创建静态库的方法略有不同。创建共享库的过程如下所述：

1. 编译目标文件时使用 GCC 的 `-fPIC` 选项，这能产生与位置无关的代码并能被加载到任何地址。
2. 使用 GCC 的 `-shared` 和 `-soname` 选项。
3. 使用 GCC 的 `-wl` 选项把参数传递给链接器 `ld`。
4. 使用 GCC 的 `-l` 选项显式地链接 C 库，以保证可以得到所需的启动（startup）代码，从而避免程序在使用不同的，可能是不兼容版本的 C 库的系统上不能启动执行。

让我们回到错误处理库，建立一个共享库，首先编译以下目标文件：

```
$ gcc -fPIC -g -c liberr.c -o liberr.o
```

然后链接库：

```
$ gcc -g -shared -Wl,-soname,liberr.so -o liberr.so.1.0.0 liberr.o  
-lc
```

由于不想把该库作为系统库安装到 `/usr/lib` 或 `/usr` 上，所以要建立必须的符号链接，一个用于 `soname`：

```
$ ln -s liberr.so.1.0.0 liberr.so.1
```

另一个是链接程序在使用 `-lerr` 链接到 `liberr` 时使用的：

```
$ ln -s liberr.so.1.0.0 liberr.so
```

现在为了使用这个新的共享库，我们再看一下上一节中的测试程序。我们需要告诉链

接程序要链接哪个库，这个库在哪里，因此就要使用-l和-L选项：

```
$ gcc -g errtest.c -o errtest -L. -lerr
```

最后为了执行这个程序，还要告诉动态链接器/加载器ld.so在哪里找到这个共享库：

```
$ LD_LIBRARY_PATH=$(pwd) ./errtest
```

正如前面指出的那样，环境变量LD_LIBRARY_PATH把它所包含的路径添加到受托库目录/lib和/usr/lib。ld.so将首先搜索环境变量指定的路径，确信找到你的库。另一种替代这种不方便的命令行方式的方法是把你的库路径添加到/etc/ld.so.conf，然后以root的身份运行ldconfig来更新高速缓冲区/etc/ld.so.cache。还有一种可行的方法是把库放进/usr/lib，建立一个到soname的符号链接，然后以root的身份运行ldconfig来更新高速缓冲文件。最后一种方法的优点是不必使用GCC的-L选项添加库搜索路径。

10.5 使用动态加载的共享对象

另一种使用共享库的方法是在运行时动态加载，不是作为在编译时链接和在运行时加载的库，而是作为完全独立的模块使用dl（dynamic loading，动态加载）接口显式地加载。你可能希望使用dl接口，因为它为编程人员和最终用户提供了更高的灵活性，同时还因为dl接口是一个使用库代码的更普遍的解决方案。

例如假定你正在编写下一个极好的图形操作和创建程序。在应用程序里采用专有的、使用方便的方法来处理图形数据。然而你希望能从每一种图形文件格式（可能有几百种）导入导出。要提供此功能，传统上是编写一个或多个本章讨论的库，来处理导入和导出多种格式。尽管这是一种模块化方法，但每次对库做修改或增补都要求用新的或改过的库重新链接应用程序。

若使用接口dl可以采用另一种方法：设计一个普遍的中性格式的接口来读、写和操作任何格式的图形文件。如果想向应用程序添加新的图形格式或修改已有的文件格式，只要编写一个新的模块来处理这种格式，然后让你的应用程序知道它的存在，也许是通过修改配置文件或把新模块放在一个预先定义的目录下（用来增加Netscape Web浏览器的功能的插件就是这种方法的变体）。要想增加新功能，用户只要获得新的模块（或插件），他们（也许是你）不需要重新编译应用程序，只要修改一个配置文件或把新模块复制到预先设定的目录中，应用程序中的现存代码就会加载新模块，瞧！现在就可以导入导出新的图形格式了。

接口dl（它把自己当作库libdl来实现）包含了用来加载、搜索和卸载共享对象的函数。要使用这些函数，只需在源代码中包含<dlfcn.h>，然后在编译命令或makefile中使用-lldl与libdl链接即可。注意不必链接你要使用的库。即使使用一个标准的共享库，也不必按常规方法使用。链接器不会知道共享对象，实际上当编译链接应用程序时，这些模块甚至可以不存在。

10.5.1 理解dl接口

dl接口提供了4个函数处理加载、使用、卸载共享对象的所有任务以及检查错误。

加载共享对象

为了加载共享对象，可以使用函数 `dlopen`，它的原型如下：

```
void *dlopen(const char *filename, int flag);
```

函数 `dlopen` 以 `flag` 指定的模式加载由 `filename` 指定的共享对象。`filename` 可以是一个绝对路径名、一个文件名或 `NULL`。如果是 `NULL`，`dlopen` 打开当前执行的文件，也就是你的应用程序；如果是一个绝对路径名，`dlopen` 就打开那个文件；如果仅仅是一个文件名，`dlopen` 以下面给定的顺序搜索下列目录，查找文件 `$LD_LIBRARY_PATH`：

```
$LD_LIBRARY_PATH, /etc/ld.so.cache, /usr/lib, and /lib.
```

参数 `flag` 可以是 `RTLD_LAZY`，表示来自被加载的对象的符号在被调用时解析，还可以是 `RTLD_NOW`，表示来自被加载对象的所有符号在函数 `dlopen` 返回前被解析。两个中的任何一个 `flag` 如果和 `RTDL_GLOBAL` 进行逻辑“或”操作会导致导出所有的符号，就像它们被直接链接一样。

函数 `dlopen` 如果找到 `filename` 就返回一个句柄，否则返回 `NULL`。

使用共享对象

在你能够使用由命令加载的库中的代码之前，你必须知道你在找什么，并且能够想方设法访问它。函数 `dlsym` 可以满足以上两点。它的原型是：

```
void *dlsym(void *handle, char *symbol);
```

`dlsym` 在加载的对象（由 `handle` 所指的共享对象）中搜索在 `symbol` 中命名的符号或函数。参数 `handle` 必须是函数 `dlopen` 返回的句柄；参数 `symbol` 是一个标准的 C 字符串。

函数 `dlsym` 返回指向符号的空指针，若发生错误则返回 `NULL`。

检查错误

正像我们在第 9 章中所讲的，强健的代码能查出尽可能多的错误并处理掉。当使用动态加载对象时，函数 `dlerror` 允许你发现更多的错误。

```
const char *dlerror(void);
```

如果任何函数出错，则函数 `dlerror` 返回一个描述错误的字符串，再把错误字符串置为 `NULL`。这样做的结果是随之而来的对函数 `dlerror` 的调用返回 `NULL`。

函数 `dlerror` 返回描述最近发生错误的字符串，在没有错误时返回 `NULL`。

卸载共享对象

当你已经使用过共享库的代码后，为了保存系统资源，特别是内存，就需要用 `dlclose` 函数卸载它。然而考虑到加载和卸载共享对象带来的时间开销，在卸载之前必须确定以后根本不再需要使用它或最近一段时间内不再需要使用它。函数 `dlclose` 关闭一个共享对象，其原型如下：

```
int dlclose(void *handle);
```


函数 `dlclose` 卸载 `handle` 所指的共享对象。此调用同时使得 `handle` 无效。由于 `dl` 库维护着动态库的链接数，它们并不被释放，只有 `dlclose` 对一个动态库调用的次数等于 `dlopen` 调用的次数时，它们所占用的资源才返回给操作系统。

10.5.2 使用 `dl` 接口

为了说明 `dl` 接口的用法，再一次回顾那个可信的出错处理库。这一次，它需要一个新的驱动程序，如程序清单 10.4 所示。

程序清单 10.4 使用 `dl` 接口

```
/*
 * dltest.c - Dynamically load liberr.so and call err_ret
 */
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int main(void)
{
    void *handle;
    void (*errfcn) (const char *fmt, ...);
    const char *errmsg;
    FILE *pf;

    /* open the library */
    handle = dlopen("./liberr.so", RTLD_NOW);
    if(handle == NULL) {
        fprintf(stderr, "Failed to load liberr.so: %s\n", dlerror());
        exit(EXIT_FAILURE);
    }

    dlerror(); /* Clear errors */
    errfcn = dlsym(handle, "err_ret"); /* Load err_ret */
    if((errmsg = dlerror()) != NULL) {
        fprintf(stderr, "Didn't find err_ret: %s\n", errmsg);
        exit(EXIT_FAILURE);
    }

    /* If we get this far, call the loaded function */
    if((pf = fopen("foobar", "r")) == NULL) {
        errfcn("Couldn't open foobar");
    }

    dlclose(handle);
    exit(EXIT_SUCCESS);
}
```

使用以下命令行编译程序清单 10.4。

```
$ gcc -g -Wall dltest.c -o dltest -ldl
```

正如你所看到的那样，`dltest` 既没有和 `liberr` 链接也没有在源代码中包含 `liberr.h` 文件。对 `liberr.so` 的访问都通过 `dl` 接口进行。除了用到的一个函数指针可能比较少见之外，程序清单 10.4 简单易懂。第二段代码示范了 `dlderror` 函数的正确用法。第一个 `dlderror` 调用把出错字符串设置为 `NULL`。调用 `dlsym` 之后，再次调用 `dlderror` 并且把它的返回值保存在另一个变量中以便于以后可以使用该字符串。程序按照调用 `err_ret` 的正常方法调用 `errfcn`。特别注意，`errfcn` 的声明要和 `err_ret` 的声明相匹配；实质上，`errfcn` 成了 `err_ret` 的别名。最后，`dltest` 卸载共享对象并退出。如果一切进行顺利的话，程序 `dltest` 就会有类似下面的输出：

```
$ LD_LIBRARY_PATH=$(pwd) ./dltest
Couldn't open foobar: No such file or directory
```

这个输出正是我们所期望的结果。把它和程序清单 10.2 的输出结果对照一下。

注意： 如果你对函数指针不熟悉，请迅速查看一下参考手册。不过简单地说，`errfcn` 的声明

```
void (*errfcn)(const char *fmt, ...);
```

表明 `errfcn` 是一个指向函数的指针，该函数有一个或多个 `const char*` 类型的参数并且什么也不返回。

10.6 小 结

本章讨论了创建和使用编程库。在简要回顾了库的兼容性和设计问题并且了解了常用库之后，你学习了怎样通过一些常用工具来使用库，以及怎样创建静态库和共享库。这一章还介绍了 `dl` 接口，它是另一种加载并使用库代码的可选方法。下一章将开始本书的第 2 部分“输入、输出、文件和目录”，这一部分将深入介绍 I/O 以及和文件系统的交互操作。

第 2 部分 输入、输出、文件和目录

第 11 章 输入和输出

本章介绍基于文件描述符的输入和输出操作 (I/O)。这种类型的文件 I/O 是 Linux 所特有的。使用标准库 I/O 函数 (例如在 `<stdio.h>` 中声明的函数) 有更好的可移植性, 特别对于非 Linux 和非 UNIX 平台更是如此, 第 12 章将介绍这类函数。

11.1 基本特点和概念

在开始使用文件处理接口之前, 需要了解文件这一思想下面所蕴含的概念和核心特性。本节在开始实际编程之前首先较详细地讨论这些思想和概念。

大多数 Linux 资源都能以文件的方式来访问。结果在 Linux 系统里有了许多种不同的文件。在一个 Linux 系统上能够出现的部分类型的文件如下:

- 普通文件
- 无名管道和有名管道
- 目录
- 设备
- 符号链接
- 套接口

普通文件 (regular file) 称为磁盘文件, 并且被定义为能够进行随机存取的数据存储单位。它们是面向字节的, 意思是从其中读出或向其写入的基本单位是单个字节, 单个字节也与单个字符相对应。当然, 既使你能够经常读出或写入多个字节, 可基本的单位却仍然是单个字符或字节。

管道 (pipe, 将在第 17 章中详细讨论) 就如同它的名字所暗示的那样——是一个从一端接受数据并把数据传向另一端的数据通道。一端执行写入操作, 而另一端执行读出操作。有两种类型的管道: 有名管道和无名管道。之所以称为无名管道 (unnamed pipe) 是因为它们出现在系统的硬盘上从来没有名称, 比如 `/home/kwall/somefile`。相反, 无名管道只是根据需要在内存中创建并在内存中消失 (严格地说, 是在内核中)。而且, 正如你在第 17 章中所看到的那样, 无名管道只通过数字而从不通过文件名来引用。然而, 你可以使用同一接口来读写无名管道, 这个接口和读写一个基于磁盘的普通文件的接口是一样的。

相反地, 有名管道 (named pipe) 拥有自己的名字。它们最常使用的场合是在两个进

程需要共享数据而又没有共享文件描述符的时候。

目录 (directory) 也称为目录文件, 它是包含了保存在目录中文件列表的简单文件。

设备文件 (device file) 也称为特殊文件 (special file), 该文件提供了到大多数物理设备的接口。它们不是字符型特殊文件就是块特殊文件。字符型特殊文件 (character special file) 一次只能读出或写入一个字节或字符的数据。字符设备的例子包括调制解调器、终端、打印机、声卡以及鼠标。另一方面, 块特殊文件 (block special file) 必须以一定大小的块来读出或写入数据 (一个块是指某种任意大小的数据块; 例如, 512 字节或 1K 字节)。块设备包括 CD-ROM 驱动器、RAM 驱动器和磁盘驱动器。一般而言, 字符设备用于传输数据, 而块设备用于存储数据。设备文件保存在 /dev 目录下。

符号链接 (symbolic link) 是包含了到达另一个文件的路径的文件。从功能上看, 它们的行为和命令的别名很相似。大多数处理文件的调用都是处理链接指向的真实文件而不是链接文件本身 (它称为跟随链接)。

套接口 (socket) 的运行更像管道, 但是它能够让处于不同机器上的进程进行通信。

但是, 无论哪种文件类型, Linux 的文件抽象 (file abstraction) ——也就是说, 它习惯于将几乎所有的东西按文件对待——能够让你使用相同的接口打开、关闭、读取和写入不同的文件。文件抽象提供给你一个一致的、统一的接口用来和所有的设备和文件类型进行交互, 从而免去你必须记住写入块设备、符号链接或目录所用的不同方法的麻烦。

文件模式

文件的模式是一个 16 比特位的域, 它由一个八进制数表示, 从而说明了文件的类型和访问权限。访问权限和它们的修饰位填满了模式的低 12 比特位。最高 4 比特位表示文件的类型。图 11.1 显示了文件的模式和它的组成元素。

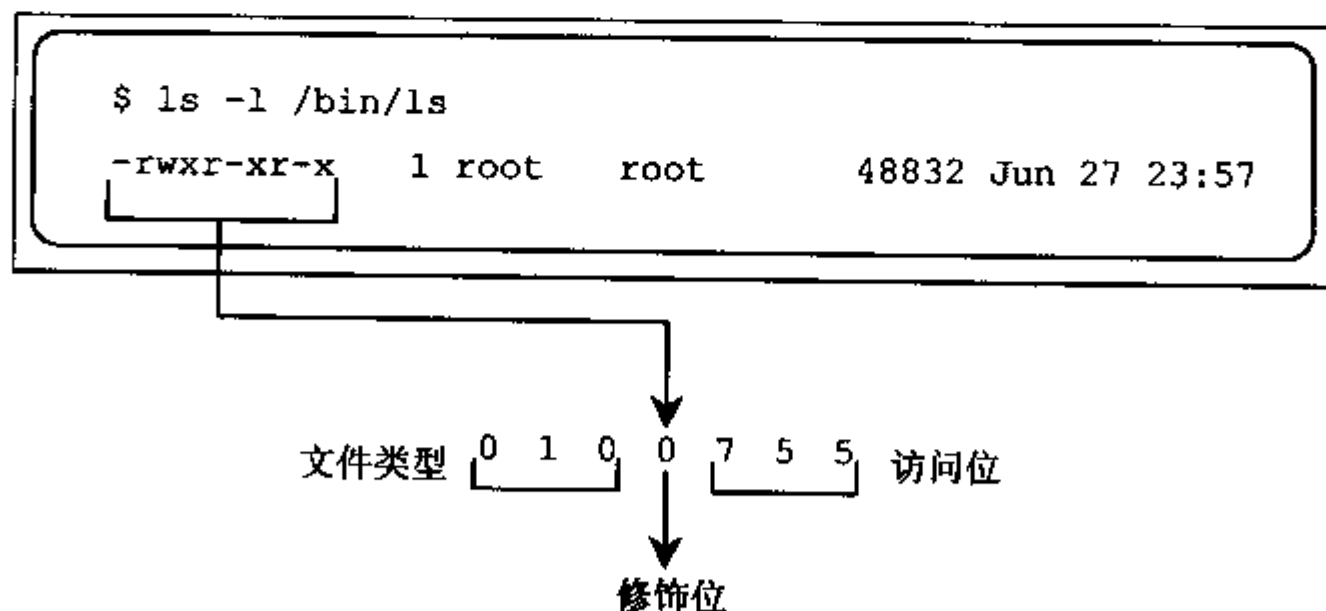


图 11.1 文件模式的组成元素

访问位

八进制数最低的 3 位说明了文件的访问权限。正如你在图 11.1 中所看到的那样, 从右向左读取八进制模式, 各比特位分别指定了其他人、本组人和所有者的访问权限。值 1 对应执行权限; 值 2 对应写权限; 值 4 对应读权限。图 11.1 中文件对于本组人和其他人是可

读和可执行的，而对于所有者 root 是可读/可写/可执行的。

修饰位

文件模式的第四位是文件模式的修饰位：它指出文件是否设置了 setUID 位、setGID 位和粘附位。正如你将在第 13 章中所学习到的那样，当一个进程执行一个设置了 setUID 位或 setGID 位的文件时，它的有效（effective）UID 或 GID 将被分别设置为该文件的所有者或所有组。当一个文件设置了粘附位时（用在第四位上的大写 S 表示），它将迫使内核尽可能长地把该文件保存在内存中（因此称为粘附），即使它不被执行也如此，因为这样做能减少执行的时间。图 11.1 中的文件没有设置修饰位，所以这个文件既没有设置 setUID 位、setGID 位，也没有粘附特性。

注意：文件的修饰位和访问权限位都是位掩码（bitmask）；也就是说，它们是能够用 C 的位操作功能，如“<<”（左移）和“~”（位补），以比特位方式操作和计算的整数。幸运的是，Linux 提供了一套宏和符号常量（见表 11.1），可以方便地对文件模式进行解码。

表 11.1 文件访问和修饰的位掩码宏

名称	掩码	描述	POSIX
S_ISUID	0004000	设置 UID 位	Yes
S_ISGID	0002000	设置 GID 位	Yes
S_ISVTX	0001000	粘附位	No
S_IRWXU	00700	用户（所有者）具有读/写/执行权限	Yes
S_IRUSR	00400	用户具有读权限	Yes
S_IWUSR	00200	用户具有写权限	Yes
S_IXUSR	00100	用户具有执行权限	Yes
S_IRWXG	00070	本组人具有读/写/执行权限	Yes
S_IRGRP	00040	本组人有读权限	Yes
S_IWGRP	00020	本组人有写权限	Yes
S_IXGRP	00010	本组人有执行权限	Yes
S_IRWXO	00007	其他人具有读/写/执行权限	Yes
S_IROTH	00004	其他人有读权限	Yes
S_IWOTH	00002	其他人有写权限	Yes
S_IXOTH	00001	其他人有执行权限	Yes

文件类型

文件类型（file type）是一个代表文件类型的简单数值。下面是 Linux 的文件类型：

- 套接口（Socket）
- 符号链接（Symbolic link）
- FIFO

- 普通文件 (Regular file)
- 目录 (Directory)
- 块设备 (Block device)
- 字符设备 (Character device)

表 11.2 用于决定文件类型的符号常量。

表 11.2 文件类型常量

名称	掩码	描述	POSIX
S_IFMT	00170000	所有文件类型的位掩码	No
S_IFSOCK	0140000	套接口文件	No
S_IFLNK	0120000	符号链接文件	No
S_IFREG	0100000	普通文件	No
S_IFBLK	0060000	块设备文件	No
S_IFDIR	0040000	目录文件	No
S_IFCHR	0020000	字符设备文件	No
S_IFIFO	0010000	FIFO 文件	No

本章 11.3.6 小节将会示范如何使用这些符号常量得出文件的类型。

本节材料可能让人有些望而生畏。放松些！虽然它是本章其余部分的基础，但此时你所需要理解的只是知道 Linux 有许多种不同的文件类型，并且你可以使用表 11.2 列出的常量判断一个文件的类型。当你学完本章的其余部分并且尝试了示例程序之后，再复习一下表 11.1 和 11.2。

提示： 要从用户而不是程序员的角度了解操作文件模式的信息，可参考 SAM 出版的“Teach Yourself Linux in 24 Hours” (作者为 Bill Ball)、“Linux Unleashed” (作者为 Tim Parker) 或者 “Special Edition Using Linux” (作者为 Jack Tackett 和 Steven Burnett) 三本书。

umask

在 11.3.1 小节里你会发现可以在创建文件和目录的同时设置它们的权限。但是，在系统和用户两个级别上，你要求的权限会被进程的 umask 修改，umask 是新创建的文件和目录应关闭的权限位的位掩码。umask 只影响文件的权限位；不能用 umask 改变修饰位和文件类型。

你可以改变进程的 umask，但只能让它更严格，而不能更宽松。完成这一功能的调用称为 umask 调用，其原型如下：

```
#include <sys/stat.h>
mode_t umask(mode_t newmask);
```

这个函数把进程的新 umask 设置为 newmask。无论调用成功与否，umask 函数都返回原来的 umask 值。程序清单 11.1 中的程序调用 umask 以设置一个更严格的 umask 掩码。

程序清单 11.1 使用 umask

```
/*
 * newmask.c - Change the umask
 */
#include <sys/stat.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    mode_t newmask = 0222, oldmask;

    system("touch before");
    oldmask = umask(newmask);
    printf("old umask is %#o\n", oldmask);
    printf("new umask is %#o\n", newmask);
    system("touch after");

    exit(EXIT_SUCCESS);
}
```

要编译这个程序，执行 `make newmask`。newmask 执行的结果如下：

```
$ ./newmask
old umask is 022
new umask is 0222
$ ls -l after before
-rw-r--r--      1 kwallusers      0 Jun 27 21:31 before
-r--r--r--      1 kwallusers      0 Jun 27 21:31 after
```

正如程序的输出所显示的那样，新的 umask 被设置为 0222。工具 touch 通常创建 644 模式（这取决于当前的 umask）的文件。但是，umask 0222 指出所有用户的一切新文件都应该是只读（0444 模式）的。结果，创建的文件 after 其模式为 444，和 ls 报告的结果完全吻合。对照 before 和 after 的权限，看看改变 umask 所产生的结果。

11.2 理解文件描述符

在介绍使用文件描述符的函数之前，特别是前面章节曾讨论过编写可移植的兼容 ANSI 的代码，说明什么是文件描述符、为什么你会想到或者需要使用它们等问题会很有帮助。本节还将讨论一个核心的 Linux I/O 概念——文件抽象。

11.2.1 文件描述符的概念

文件描述符是个很小的正整数，它是一个索引值，指向内核为每一个进程所维护的该进程打开文件的记录表。例如，每个进程启动时都打开 3 个文件：标准输入、标准输出和

标准出错文件（本书使用它们的缩写 `stdin`、`stdout` 和 `stderr`）。这 3 个文件分别对应于文件描述符 0、1 和 2。实际上，你可能已经用过基于描述符的 I/O 操作，但却没有意识到这一点。当你把命令的输出重定向到 `/dev/null` 时，所使用的方法如下：

```
$ egrep void *.c 2> /dev/null
```

这表明你正在使用文件描述符。在重定向符号前面的数字 2 就代表了 `stderr`。类似地，下面的命令使用 `stdout` 和 `stderr` 两个描述符：

```
$ egrep void *.c 2 > &1 egrep.out
```

在这里，正如你可能知道的那样，`stderr`（文件描述符 2）被附加到了 `stdout`（文件描述符 1）后面，而接下来又被重定向输出到文件 `egrep.out` 中。

提示： 应该使用 `<unistd.h>` 中定义的 3 个宏来代替数字 0、1 和 2：`STDIN_FILENO`、`STDOUT_FILENO` 和 `STDERR_FILENO`。我非常鼓励你使用这些宏而不是数字 0、1 或 2，因为你的程序可能会在一个 `stdin`、`stdout` 和 `stderr` 不与整数 0、1、2 相对应的系统上进行编译。

11.2.2 文件描述符的优缺点

基于描述符的 I/O 操作最主要的缺点就是它不能移植到 UNIX 以外的系统上去。尽管大多数操作系统都有输入和输出的概念，但是它们可能不会以 Linux 那样的方式处理 I/O。如果你要把一个程序移植到一个非 Linux 的环境中，那么应该使用 C 标准所定义的 I/O 功能。

基于描述符的 I/O 操作的第二个缺点是这种方式不太直观。代码中间散布着神奇数字（例如 0、1 和 2），使得代码很难阅读。虽然对此习惯了的 Linux 程序员会对这些惯例很熟悉，但是刚刚接触 Linux 编程的新手以及各种新程序员可能会发现这样的代码简直不知所云。解决问题的一种方法是使用上面提示中介绍的 `STDIN_FILENO`、`STDOUT_FILENO` 和 `STDERR_FILENO` 宏。你会在本章的后面看到其用法的示例程序示范。

基于文件描述符的 I/O 操作的优点是这种方式兼容 POSIX 标准，而且在有些情况下，这是能够实现某些 I/O 操作的唯一途径。Linux 紧密遵循的 POSIX 标准中充满了使用文件描述符进行输入和输出的接口，所以只要程序不需要移植到非 POSIX 平台，如 Windows 9x 上，那么使用这类 I/O 操作就没问题。而且，基于描述符的 I/O 是合乎习惯的用法；也就是说，在 Linux 和 UNIX 程序中经常会用到。

更重要的是，许多 Linux 和 UNIX 系统调用都依赖于文件描述符。比如，低级的 `open`、`close`、`read` 和 `write` 调用都使用文件描述符。实际上，在某些情况下，比如本书第 19 章到第 21 章讨论的 TCP/IP 套接口编程接口，就只能通过文件描述符执行输入和输出操作。

在 Linux 上，几乎每一样东西都是一个文件，至少抽象地看是这样。这一事实也是 Linux 最具独创性的设计特色之一，因为它让大量的资源，比如内存、磁盘空间、进程间通信通道、网络通信通道、磁带驱动器、控制台、串口、伪终端、打印端口、声卡、鼠标甚至其他运行着的进程具有了统一的编程接口。

11.3 使用文件描述符

如前所述，许多系统调用要使用文件描述符。本节对每一个这样的调用作简要描述并给出示例。特别是，你将会学到怎样使用 `open`、`create`、`close`、`read`、`write`、`ftruncate`、`lseek`、`fsync`、`fstat`、`fchown`、`fchmod`、`flock`、`fcntl`、`dup`、`dup2`、`select` 和 `ioctl`。

提示： 你注意有些操作文件描述符的函数名字以“f”开头吗？这可不只是因为偶然。通常，以“f”开头的系统调用使用文件描述符，而它们的对应体（名字没有开头的“f”），比如 `truncate`、`chmod`、`chown` 和 `stat` 则使用文件指针，文件指针是 C 标准 I/O 库的一部分。

11.3.1 打开关闭文件描述符

有两个调用都能打开文件，即 `open` 和 `creat`。使用它们需包含头文件 `<sys/types.h>`、`<sys/stat.h>` 和 `<fcntl.h>`。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int creat(const char *pathname, int flags, mode_t mode);
```

`open` 试图打开目录 `pathname` 中的一个文件，`flag` 指定访问该文件的方式。参数 `mode` 包含了文件在创建时的模式。必须把 `flags` 设置为 `O_RDONLY`、`O_WRONLY` 或 `O_RDWR`，它们分别表示只读、只写或读/写访问。另外，你可以设置零个或多个表 11.3 列出的值（如果使用了多个值，则必须按位“或”）。如果默认的文件模式（`umask`）满足你的要求，则使用 `open` 的第一种形式。如果你希望像使用进程的 `umask` 那样也设置一个特定的文件模式，则使用 `open` 的第二种形式。两种形式的 `open` 成功后都返回一个文件描述符。如果失败，则返回 -1 并且设置 `errno` 变量。

表 11.3 系统调用 `open` 使用的标志

标志	说明
<code>O_RDONLY</code>	只读访问打开文件
<code>O_WRONLY</code>	只写访问打开文件
<code>O_RDWR</code>	读和写访问打开文件
<code>O_CREAT</code>	如果文件不存在则建立文件
<code>O_EXCL</code>	仅与 <code>O_CREAT</code> 连用，如果文件已存在，则强制 <code>open</code> 失败
<code>O_NOCTTY</code>	如果打开的文件是一个终端，就不会成为打开其进程的控制终端
<code>O_TRUNC</code>	如果文件存在，则将文件的长度截至 0
<code>O_APPEND</code>	将文件指针设置到文件的结束处（如果打开来写）
<code>O_NONBLOCK</code>	如果读操作没有 blocking（由于某种原因被拖延）则无法完成时，读操作返回 0 字节

(续表)

标志	说明
O_NDELAY	同 O_NONBLOCK
O_SYNC	在数据被物理地写入磁盘或其他设备之后操作才返回

`creat` 也能打开一个文件，如果该文件不存在，则创建它。和 `open` 一样，`creat` 也在调用成功后返回一个文件描述符，或者如果失败，则设置 `errno` 变量并返回-1。`creat` 的原型为

```
int creat(const char *pathname, mode_t mode);
```

它等价于

```
open(pathname, O_CREAT | O_TRUNC | O_WRONLY, mode);
```

本书中的程序没有使用 `creat` 的原因有两个。第一，`creat` 拼写有错误。第二，调用 `open` 更常见，而且使用 `open` 的 `O_CREAT` 标志和直接使用 `creat` 取得结果是一样的。如果调用 `creat` 成功，则返回一个文件描述符；如果调用失败，则设置 `errno` 变量并返回-1。

为了在使用完某个文件后关闭它，可以采用系统调用 `close`。`close` 只有一个参数，即 `open` 返回的文件描述符。`close` 的原型为

```
#include <unistd.h>
int close(int fd);
```

一旦调用了 `close`，则该进程对文件所加的锁全都被释放，即使这些锁是通过别的文件描述符加上的。如果要被关闭的文件导致它的链接（硬链接或符号链接到该文件的链接数目）数为 0，则该文件会被删除。如果这是和一个打开的文件相关联的最后（或惟一）的文件描述符，则释放打开文件表中对应该文件的项。程序清单 11.2 中的 `hello` 程序示范了打开和关闭某个文件的操作。

程序清单 11.2 使用 `open` 和 `close`

```
/*
 * fdopen.c - Opening and closing file descriptors
 */
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int fd;
    char path[] = "hello";

    if((fd = open(path, O_CREAT | O_TRUNC | O_WRONLY, 0644)) < 0){
```

```

        perror("open");
        exit(EXIT_FAILURE);
    }else{
        printf("opened %s\n",path);
        printf("descriptor is %d\n",fd);
    }

    if(close(fd) < 0) {
        perror("close");
        exit(EXIT_FAILURE);
    } else {
        printf("closed %s\n",path);
    }

    exit(EXIT_SUCCESS);
}

```

编译该程序的命令为 `make fdopen`。执行该程序的输出结果为

```

$ ./fdopen
opened hello
descriptor is 3
closed hello

```

`open` 语句试图以只读模式打开 `hello` 文件。如果该文件不存在,则 `O_CREAT` 会创建它,而如果该文件存在, `O_TRUNC` 将该文件的长度设置为 0,就好像它是新创建出来的一样。打开该文件后, `fdopen` 又提示关闭它。

需要特别注意的是,检查 `close` 返回值的源代码。虽然通常不这样做,但是不检查 `close` 返回值是一个严重的编程错误,原因有二。首先,在网络文件系统中,例如 NFS, `close` 调用会因为网络延迟而失败。其次,许多系统都配置有写后缓冲 (`write-behind caching`) 的作用,这意味着即使 `write` 调用成功返回,操作系统也要等到一个更方便的时候执行实际的磁盘写入操作。正如 `close(2)` 手册页面所叙述的:

“错误状态可能会在写入操作结束后晚些时间才报告,但肯定会在关闭文件时报告。在关闭文件时不检查返回值可能导致在不知情的情况下丢失数据。”

11.3.2 读写文件描述符

系统调用 `read` 用于从文件描述符对应的文件中读取数据。它的原型如下:

```

#include <unistd.h>
ssize_t read(int fd, const void *buf, size_t count);

```

`fd` 必须是以前的 `open` 调用返回的有效文件描述符。`buf` 指定存储读出数据的缓冲区,而 `count` 指定读出的字节数。如果调用成功 `read` 返回读出的字节数,如果出错则返回-1 并设置 `errno` 变量。如果遇到 EOF (end of file, 文件末尾), `read` 返回 0。

系统调用 `write` 用于向文件描述符对应的文件写入数据。

```

#include <unistd.h>

```

```
ssize_t write(int fd, const void *buf, size_t count);
```

fd 是以前的 open 调用返回的有效文件描述符。buf 是指向保存写入数据的缓冲区的指针，而 count 指定写入的字节数。如果调用成功 write 返回写入的字节数。如果调用失败则返回-1 并设置 errno 变量。

程序清单 11.3 说明了如何使用读和写。

程序清单 11.3 使用 read 和 write

```
/*
 * fdread.c - The read and write system calls
 */
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int fdsrc, fdnull, fdtmp, numbytes;
    int flags = O_CREAT | O_TRUNC | O_WRONLY;
    char buf[10];

    /* open the source file, /dev/null, and /tmp/foo.bar */
    if((fdsrc = open("fdread.c", O_RDONLY, 0644)) < 0) {
        perror("open fdread.c");
        exit(EXIT_FAILURE);
    }
    if((fdnull = open("/dev/null", O_WRONLY)) < 0) {
        perror("open /dev/null");
        close(fdsrc); /* close this since we've opened it */
        exit(EXIT_FAILURE);
    }
    if((fdtmp = open("/tmp/foo.bar", flags, 0644)) < 0) {
        perror("open /tmp/foo.bar");
        close(fdsrc); /* have to close both of these now */
        close(fdnull);
        exit(EXIT_FAILURE);
    }

    /* read and write 10 bytes at a time */
    while((numbytes = read(fdsrc, buf, 10)) > 0) {
        if(write(fdnull, buf, 10) < 0) {
            perror("write /dev/null");
        }
        if(write(fdtmp, buf, numbytes) < 0) {
            perror("write /tmp/foo.bar");
        }
    }
}
```

```

    }
}

/* close files and exit */
close(fdsrcc);
close(fdnull);
close(fdtmp);

exit(EXIT_SUCCESS);
}

```

程序打开 3 个文件，一个用于读取数据，两个用来写入数据（执行 `make fdread` 编译它）。文件 `/tmp/foo.bar` 相对来说没有太多令人感兴趣的地方，但是要注意程序打开了设备文件 `/dev/null`，就好像它是个普通文件一样。因此，你应该看到你能够把大多数设备当作文件。用于普通文件的文件处理语义也同样适用于设备文件和其他特殊文件。该程序的另一个特点是，当执行向磁盘文件 `/tmp/foo.bar` 写入的操作时，程序只写入至多 `numbytes` 个字符，这样就避免了向文件末尾写入空白字节。在达到文件末尾时最后执行的 `read` 操作不会读满 10 个字节，但是 `write` 操作会尽可能多地写入要求它写入的字节。通过要求写入 `numbytes` 个字节，程序不会在文件末尾附加额外的字符。最后注意，`read` 调用检查返回值是否大于 0——这样做会发现读操作出错的情况，此时返回 -1，而且也能发现读到文件末尾的情况，此时返回 0。

11.3.3 使用 `ftruncate` 缩短文件

系统调用 `ftruncate` 把文件描述符 `fd` 引用的文件缩短到 `length` 指定的长度。

```

#include <unistd.h>
int ftruncate(int fd, off_t length);

```

`ftruncate` 成功时返回 0。如果出错返回 -1 并设置 `errno` 变量。

11.3.4 使用 `lseek` 定位文件指针

函数 `lseek` 在用描述符 `fd` 打开的文件里把文件指针设定到相对于 `whence` 值偏移 `offset` 的位置，文件指针是文件中执行读写操作的位置。

```

#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);

```

`whence` 可以是这里列出的 3 个常量中的一个：

- `SEEK_SET` 设置文件指针到文件内 `offset` 字节处。
- `SEEK_CUR` 设置指针的位置相对于指针当前位置向前 `offset` 字节处。`offset` 可以为负。
- `SEEK_END` 设置指针的位置为从文件结尾往回 `offset` 字节处。

如果调用成功 `lseek` 返回新指针的位置，如果出错则返回 -1 (`off_t`)，并且会恰当地设置 `errno` 变量。

11.3.5 使用 fsync 同步到硬盘

系统调用 `fsync` 将所有已写入文件描述符 `fd` 的数据真正地写到磁盘或其他下层设备上。

```
#include <unistd.h>
int fsync(int fd);
#ifdef _POSIX_SYNCHRONIZED_IO
    int fdatasync(int fd);
#endif
```

Linux 文件系统可以使数据在写入磁盘前先在内存中保留几秒钟，以此更高效地处理磁盘 I/O。如果调用成功 `fsync` 返回 0；否则，返回 -1 并设置 `errno` 变量。

注意： `fdatasync` 调用类似于 `fsync`，但是不写入文件的索引节点 (inode) 信息，如修改时间等。

11.3.6 使用 fstat 获得文件信息

系统调用 `fstat` 返回文件描述符 `fd` 引用的文件的相关信息，并把结果保存在 `buf` 指向的结构 `struct stat` 中。通常如果调用成功则返回 0。如果调用失败，`fstat` 返回 -1 并设置 `errno` 变量。

```
#include <sys/stat.h>
#include <unistd.h>
int fstat(int fd, struct stat *buf);
```

这里是援引自手册页面的 `struct stat` 定义：

```
struct stat
{
    dev_t      st_dev;      /* device */
    ino_t      st_ino;      /* inode */
    mode_t     st_mode;     /* protection */
    nlink_t    st_nlink;    /* number of hard links */
    uid_t      st_uid;      /* user ID of owner */
    gid_t      st_gid;      /* group ID of owner */
    dev_t      st_rdev;     /* device type (if inode device) */
    off_t      st_size;     /* total size, in bytes */
    unsigned long st_blksize; /* blocksize for filesystem I/O */
    unsigned long st_blocks; /* number of blocks allocated */
    time_t     st_atime;    /* time of last access */
    time_t     st_mtime;    /* time of last modification */
    time_t     st_ctime;    /* time of last change */
};
```

为了正确解释文件类型，有一套宏能够计算 `stat` 结构的 `st_mode` 成员。表 11.4 列出了这些宏。

表 11.4 文件类型宏

宏	描述
S_ISLNK(mode)	如果文件是一个符号链接，则返回真
S_ISREG(mode)	如果文件是一个普通文件，则返回真
S_ISDIR(mode)	如果文件是一个目录，则返回真
S_ISCHR(mode)	如果文件是一个字符设备，则返回真
S_ISBLK(mode)	如果文件是一个块设备，则返回真
S_ISFIFO(mode)	如果文件是一个 FIFO，则返回真
S_ISSOCK(mode)	如果文件是一个套接口，则返回真

要使用这些宏，可以把 stat 结构的成员 st_mode 作为所列宏的 mode 参数。程序清单 11.4 示范了宏的用法以及一般如何使用 fstat 的例程。程序清单 11.5 显示了如何使用 ftruncate、lseek 和 fsync。

程序清单 11.4 使用 fstat

```

/*
 * mstat.c - Naïve stat(1) program
 */
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    struct stat buf;
    mode_t mode;
    char type[80];
    int fd;

    /* validate the command line */
    if(argc != 2) {
        puts("USAGE: mstat {file}");
        exit(EXIT_FAILURE);
    }

    /* open the file */
    if((fd = open(argv[1], O_RDONLY)) < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    /* get file stats */

```

```

if((fstat(fd, &buf)) < 0) {
    perror("fstat");
    exit(EXIT_FAILURE);
}
mode = buf.st_mode;
printf("    FILE: %s\n", argv[1]);
printf("    INODE: %ld\n", buf.st_ino);
printf("    DEVICE: %d,%d\n", major(buf.st_dev),
        minor(buf.st_dev));
printf("    MODE: %#o\n", mode & ~(S_IFMT));
printf("    LINKS: %d\n", buf.st_nlink);
printf("    UID: %d\n", buf.st_uid);
printf("    GID: %d\n", buf.st_gid);
if(S_ISLNK(mode))
    strcpy(type, "Symbolic line");
else if(S_ISREG(mode))
    strcpy(type, "Regular file");
else if(S_ISDIR(mode))
    strcpy(type, "Directory");
else if(S_ISCHR(mode))
    strcpy(type, "Character device");
else if(S_ISBLK(mode))
    strcpy(type, "Block device");
else if(S_ISFIFO(mode))
    strcpy(type, "FIFO");
else if(S_ISSOCK(mode))
    strcpy(type, "Socket");
else
    strcpy(type, "Unknown type");
printf("    TYPE: %s\n", type);
printf("    SIZE: %ld\n", buf.st_size);
printf("BLK SIZE: %ld\n", buf.st_blksize);
printf("    BLOCKS: %d\n", (int)buf.st_blocks);
printf("ACCESSED: %s", ctime(&buf.st_atime));
printf("MODIFIED: %s", ctime(&buf.st_mtime));
printf("    CHANGED: %s", ctime(&buf.st_ctime));

/* close the file */
if(close(fd) < 0) {
    perror("close");
    exit(EXIT_FAILURE);
}

exit(EXIT_SUCCESS);
}

```

执行 `make mstat` 编译这个程序。为运行 `mstat`，把你感兴趣的文件名作为参数传递给它。运行结果如下：


```

$ ./mstat mstat.c
  FILE:  mstat.c
  INODE:  5168
  DEVICE:  3,1
  MODE:  0644
  LINKS:  1
  UID:  500
  GID:  100
  TYPE:  Regular file
  SIZE:  1755
  BLK SIZE:  4096
  BLOCKS:  4
  ACCESSED:  Web Jun 28 01:25:07 2000
  MODIFIED:  Web Jun 28 00:49:04 2000
  CHANGED:  Web Jun 28 00:49:04 2000

```

显然这里的代码写得很难看，但是却示范了如何使用 `fstat` 函数。当取得文件的信息后，程序显示出 `stat` 结构每个成员的值。当它显示文件类型时，程序尽很大努力把无意义的数字转换为可读的形式，因此使用了 `if...else if` 结构（应该在函数中避免这样的用法）。`mstat` 使用表 11.2 介绍的常量 `S_IFMT` 掩盖文件模式中的文件类型比特位，从而显示只包含权限位和修饰位的文件模式。

这段代码还使用了 `ctime` 函数把 `atime`、`mtime` 和 `ctime` 值转换成易于理解的字符串。该程序虽然有些粗糙但确实显示了可能实现什么样的功能，并且可以作为进一步工作的良好出发点。特别地，还可以添加代码来检验作为参数传递的文件名是否有效。

程序清单 11.5 使用 `ftruncate`、`lseek` 和 `fsync`

```

/*
 * seek.c - Using lseek, fsync, and ftruncate
 */
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char ftmp[ ] = "tmpXXXXXX";
    char buf[10];
    struct stat statbuf;
    int i, infd, outfd;

    /* open the input file */
    if((infd = open("devices.txt", O_RDONLY)) < 0) {
        perror("open devices.txt");
    }

```

```

        exit(EXIT_FAILURE);
    }
    /* create a temporary file for output */
    if((outfd = mkstemp(ftmp)) < 0) {
        perror("mkstemp");
        exit(EXIT_FAILURE);
    }
    printf("output file is %s\n",ftmp);

    /* set the initial location in the file */
    lseek(infd, 100, SEEK_SET);
    /*
     * copy the first ten out of every 100 bytes
     * to the output file
     */
    for(i = 0; i < 10; ++i) {
        read(infd, buf, 10);
        write(outfd, buf, 10);
        lseek(infd, 90, SEEK_CUR);    /* jump forward 90 bytes */
    }

    /* show size before and after ftruncate */
    fstat(outfd, &statbuf);
    printf("before ftruncate, %s is %ld bytes\n", ftmp, statbuf.
           st_size);
    ftruncate(outfd, statbuf.st_size / 2);
    fsync(outfd);
    fstat(outfd, &statbuf);
    printf("after ftruncate, %s is %ld bytes\n", ftmp, statbuf.
           st_size);

    /* close 'em up and get outta here */
    close(infd);
    close(outfd);
    exit(EXIT_SUCCESS);
}

```

这个程序（用 `make seek` 编译）完成两项功能。首先，它在其输入文件 `devices.txt` 中多次定位然后读取 10 个字节数据。该程序使用 `mkstemp` 调用，而不是在程序中指定文件名来打开当前目录下名称唯一的文件。在把文件指针设置在文件内部自开头起 100 个字节后，程序读取 10 个字节的数据，并写入到输出文件中，然后向前移动 90 字节。输出文件的名字每次执行都不同。其次，程序在调用 `ftruncate` 的前后使用 `fstat` 调用显示临时文件的长度，该长度以字节为单位。运行的实例表明 `ftruncate` 调用能够工作：

```

$ ./seek
output file is tempzU6TDn
before ftruncate, tmpzU6TDn is 100 bytes
after ftruncate, tmpzU6TDn is 50 bytes

```

正如你所看到的那样，临时文件的长度从 100 字节缩短到 50 字节。

11.3.7 使用 fchown 改变文件所有权

系统调用 `fchown` 是程序员使用的 `chown` 命令的等价体，它能让你改变与打开文件相关联的所有者和所有组。它的原型为

```
#include <sys/types.h>
#include <unistd.h>
int fchown(int fd, uid_t owner, gid_t group);
```

正如你能预料到的那样，`fd` 是要操作的文件的文件描述符，`owner` 是新的所有者的数字 UID（用户 ID），而 `group` 是新的 GID（所有组 ID）。`owner` 或 `group` 的值为 -1 将不改变其值。通常，如果成功，`fchown` 返回 0；如果失败，则返回 -1，并设置 `errno` 变量。

注意： 一个普通用户可以将文件的所有组改变为其所属组之一。只有根用户可以将所有者放到任何组中。

11.3.8 使用 fchmod 改变文件读写权

`fchmod` 调用把 `fd` 引用的文件的权限位（文件模式）改为 `mode` 指定的八进制模式。

```
#include <sys/types.h>
#include <sys/stat.h>
int fchmod(int fd, mode_t mode);
```

文件的模式经常以八进制方式来表示，每一个八进制数由 3 位组成。采用八进制而不是十六进制的原因是一些系统不能打印十六进制数中的 A~F。要记住的一点是在 C 语言中任何一个以 0 开头的数都被认为是八进制数，这也是 C 语言中不太令人满意的地方。`fchmod` 调用成功时返回 0，失败时返回 -1。失败时还要设置 `errno` 变量。参见表 11.1 中列出的文件模式和它们的八进制数值。

注意： 在某些情况下，当执行该调用或文件被修改时，内核可能会悄悄修改这些权限位以保证安全。特别是在写文件时要重置 `setuid` 和 `setgid` 位。

11.3.9 使用 flock 和 fcntl 给文件上锁

文件上锁是能够让多个进程安全、合理并按预料同时访问同一文件的方法。虽然文件上锁的目的更多是为了限制对文件的访问而不是正确进行 I/O 操作，但是本章还会保留 I/O 方面的内容，因为大多数程序员都会发现上锁的方法在 I/O 相关的代码上下文中要比在资源和进程控制的代码上下文中（这将在第三部分详细介绍）中更有用得多。

每个对某文件上锁的进程都是为了防止其他使用该文件的进程改变文件的数据，也就是说，防止由于其他进程的 I/O 操作引起不能预计的状态变化。没有什么固有的原因说两个进程不能同时从同一个文件中读取数据，但是设想一下，如果两个进程同时对一个文件进行写入操作所引起的混乱吧。它们可能会相互覆盖对方的数据，或者有时候干脆破坏掉整个文件。

系统调用 `flock` 请求或删除由文件描述符 `fd` 引用的文件上的一个建议性锁。

```
#include <sys/file.h>
int flock(int fd, int operation);
```

第二个参数 `operation` 值为 `LOCK_SH` 时，代表共享性锁；为 `LOCK_EX` 时，代表排斥性锁；为 `LOCK_UN` 时，代表解锁；值 `LOCK_NB` 可以和其他任何值进行“或”操作以防止上锁。在任何特定时刻，在一个文件上只能有一个进程施加排斥性锁，但是可以有多个进程施加共享性锁。只有当一个程序试图施加它自己的锁时，锁才会起作用；而没有尝试对文件上锁的程序仍然能够访问该文件。因此，锁只能在协同工作的程序间起作用。该调用成功时，返回值为 0；失败时返回 -1。

在 Linux 以及许多其他 UNIX 类型的系统上，存在许多种类的文件锁，它们中某些锁可以互操作，而某些锁又不能互操作。由 `flock` 施加的锁不能和由 `fcntl` 或 `lockf` 施加的锁进行通信，也不能和 `/var/lock` 下的 UUCP 锁文件进行通信。如果 Linux 内核支持，则 Linux 还实现了针对某些特殊文件的强制性锁，这些文件设置了 `setgid` 位，但却没有组可执行权；在这种情况下，用 `fcntl` 或 `lockf` 施加的锁是强制性锁。

如果要访问一个上锁文件，则一般的访问步骤如下：

1. 检查是否有锁。
2. 如果文件没有锁，则建立自己的锁。
3. 打开文件。
4. 对文件做必要的处理。
5. 关闭文件。
6. 对文件解锁。

要注意的是，进程在开始任何 I/O 操作前如何去处理锁，在对文件解锁之前如何完成所有的操作。这一过程保证了程序所有的处理操作都不会被其他进程中断。如果你在设立锁之前打开文件，或者在读取该锁之后关闭文件，另一个进程就有可能在上锁/解锁操作和打开/关闭操作之间的几分之一秒时间里访问该文件。

如果文件被上锁，你必须做出决定。许多文件 I/O 操作只花费至多几秒钟时间。你既可以选择等候几秒钟（可能使用 `sleep` 调用然后再尝试），也可以选择放弃并且向用户报告说运行的程序不能打开文件，因为另外一个进程正在使用它。

文件锁有两种类型：建议性锁和强制性锁。建议性锁（`advisory lock`）也称为合作性锁（`cooperative lock`），它依赖于这样的约定，每个使用上锁文件的进程都要检查是否有锁存在，并且尊重已有的锁。内核和系统总体上都坚持不使用建议性锁；它们都依靠程序员去遵守该约定。另一方面，强制性锁（`mandatory lock`）是由内核所执行的。当一个文件被上锁以进行写入操作的时候，在锁定该文件的进程释放该锁之前，内核会阻止任何对该文件的读或写访问。但是，采用强制性锁对性能的影响很大，因为每次 `read` 或 `write` 操作都必须检查是否存在锁。

正如有两种文件上锁类型一样，也有两种实现上锁的方法：锁定文件和记录锁定。用于实现文件上锁的两个系统调用为 `flock` 和 `fcntl`。`flock` 用于向文件施加建议性锁，而 `fcntl` 和它的包裹函数 `lockf` 既能向文件施加建议性锁也能施加强制性锁。因为系统调用 `fcntl` 符合 POSIX 标准，本章的讨论将集中在 `fcntl` 上，它能够施加建议性和强制性两种锁。

为什么集中讨论 `fcntl` 呢？主要因为它比 `flock` 更通用。`fcntl` 能够用于建立记录锁(record lock)，记录锁是对于文件一部分而不是整个文件的锁。这种对上锁行为更为细致的控制使得进程能够更好地协作以共享文件资源。更重要的是，`fcntl` 能够用于读取锁和写入锁。读取锁(read lock)也称为共享锁(shared lock)，因为多个进程能够在文件的同一部分上建立读取锁。另一方面，写入锁(write lock)经常被称为排斥锁(exclusive lock)，因为任何时刻只能有一个进程在文件的某个部分上建立写入锁。自然地，如果一个进程在文件的某些部分上建立了写入锁，那么就不能在相同的部分再建立读取锁。这是一种敏感的限制——设想一下试着读取一个内容在不断变化的文件所造成的混乱局面吧。类似地，写入锁也不能再建立在文件中已经建立了读取锁的部分上。

注意： `fcntl` 的用处不仅仅是给文件上锁。如表 11.5 所示，它还能用来控制拥有某个文件的进程组、改变和某个文件相关联的文件描述符属性以及复制文件描述符。`fcntl` 的手册页面讨论了它的这些用法。

`fcntl` 的原型如下（它有 3 种原型）：

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

一般而言，`fcntl` 改变和文件描述符 `fd` 相关联的属性。参数 `cmd` 控制 `fcntl` 做什么——`cmd` 可能的取值在表 11.5 中列出。

表 11.5 `fcntl` 的命令

命令	说明
<code>F_DUPFD</code>	复制文件描述符 <code>fd</code>
<code>F_GETFD</code>	获得 <code>fd</code> 的 <code>close-on-exec</code> 标志。如果标志没有设置，仍为 0，则文件经过 <code>exec</code> 系列调用之后仍然保持打开状态
<code>F_SETFD</code>	设置 <code>close-on-exec</code> 标志以便在 <code>arg</code> 中传送的值
<code>F_GETFL</code>	得到 <code>open</code> 设置的标志
<code>F_SETFL</code>	改变 <code>open</code> 设置的标志
<code>F_GETLK</code>	得到离散的文件锁
<code>F_SETLK</code>	设置获得离散的文件锁，不等待
<code>F_SETLKW</code>	设置获得离散的文件锁，在需要时，等待
<code>F_GETOWN</code>	检索将收到 <code>SIGIO</code> 和 <code>SIGURG</code> 信号的进程 ID 或进程组号
<code>F_SETOWN</code>	设置进程 ID 或进程组号

前面已经提到过，本节集中讨论使用 `fcntl` 设置文件锁。要设置锁，需传递值为 `F_SETLK` 或 `F_SETLKW` 的参数，设置 `lock.l_type` 的值为 `F_RDLCK`（用于读取锁）或 `F_WRLCK`（用于写入锁）。要清除锁，则设置 `lock.l_type` 的值为 `F_UNLCK`。无论设置或是清除锁，如果操作成功，`fcntl` 都返回 0。如果不能设置锁，则返回 -1，并且设置变量 `errno` 的值为 `EACCES` 或 `EAGAIN`。

要检查锁的状态, 可以使用 `F_GETLK`。如果另一个进程已经设置了锁, 则会在 `flock` * 的结构中填满相关的信息; 否则, `lock` 的成员 `l_type` 将被设置为 `F_UNLCK`, 以表明没有设置任何锁。程序清单 11.6 示范了如何通过 `fcntl` 使用文件锁。

程序清单 11.6 用 `fcntl` 进行文件锁操作

```
/*
 * lockit.c - Set file locks on a file
 */
#include <unistd.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>

/* Sets lock of type on descriptor fd */
void setlock(int fd, int type);

int main(int argc, char *argv[])
{
    int fd;

    /* Open the file */
    fd = open(argv[1], O_RDWR | O_CREAT, 0666);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    /* Set read lock */
    setlock(fd, F_RDLCK);
    printf("PID %d unlocked %s\n", getpid(), argv[1]);
    getchar();

    /* Unlock */
    setlock(fd, F_UNLCK);
    printf("PID %d unlocked %s\n", getpid(), argv[1]);
    getchar();

    /* Set write lock */
    setlock(fd, F_WRLCK);
    printf("PID %d write locked %s\n", getpid(), argv[1]);
    getchar();
    close(fd);

    exit(EXIT_SUCCESS);
}

void setlock(int fd, int type)
{
    struct flock lock;
```

```
char msg[80];

/* Describe the lock we want */
lock.l_whence = SEEK_SET;
lock.l_start = 0;
lock.l_len = 1; /* Lock a single byte */

while(1){
    lock.l_type = type;
    /* Set the lock and return to caller */
    if((fcntl(fd, F_SETLK, &lock)) == 0)
        return;

    /* Find out why we couldn't set the lock */
    fcntl(fd, F_GETLK, &lock);
    if(lock.l_type != F_UNLCK) {
        switch(lock.l_type) {
            case(F_RDLCK);
                sprintf(msg, "read lock already set by %d\n",
                        lock.l_pid);
                break;
            case(F_WRLCK);
                sprintf(msg, "write lock already set by %d\n",
                        lock.l_pid);
                break;
        }
        puts(msg);
        getchar();
    }
}
```

执行 `make lockit` 编译这个程序。为了方便，`lockit` 定义了函数 `setlock` 来处理锁操作。`while` 循环的目的是为了连续尝试设置锁，直至成功。如果第一次调用就成功了，那么它会立即返回 `main` 函数。否则，执行第二个 `if` 代码块找出无法设置锁的原因。`setlock` 函数的末尾调用 `getchar` 能够让你一次一步地执行程序。`main` 函数相当简单。首先，它打开要加锁的文件，然后尝试设置一个读取锁。第二步是给文件解锁，然后又尝试设置一个写入锁。没有特意去清除写入锁，因为当文件被关闭时内核会释放写入锁。如图 11.2 所示，在两个终端窗口中运行这个程序有助于了解 POSIX 锁是如何工作的。

```

lockit 1
[wallflower 11:18] lockit tmp/foo
PID 5374 read locked tmp/foo
PID 5374 unlocked tmp/foo
read lock already set by 5375
PID 5374 write locked tmp/foo
wallflower 11:18

lockit 2
[wallflower 11:18] lockit tmp/foo
PID 5375 read locked tmp/foo
PID 5375 unlocked tmp/foo
write lock already set by 5374
PID 5375 write locked tmp/foo
[wallflower 11:18]

```

图 11.2 在两个终端窗口中运行 lockit

首先在第一个窗口中启动 lockit (窗口标题为“lockit 1”), 然后在第二个窗口中再运行 lockit (窗口标题为“lockit 2”). 正如你所看到的, 每个进程都成功地在文件/tmp/foo 的第一个字节上建立了读取锁。但是, 第一个进程 (lockit 1 中显示其进程号为 5374) 不能设置写入锁, 直到第二个进程 (lockit 2 中显示其进程号为 5375) 释放它的读取锁为止。可以预见, 当进程号为 5374 的进程建立它的写入锁后, 进程 5375 就不能做类似的工作, 直至 5374 终止并释放掉它的锁为止。

11.3.10 使用 dup 和 dup2 调用

系统调用 dup 和 dup2 能够复制文件描述符。dup 返回新的文件描述符 (没使用的文件描述符最小的编号)。dup2 可以让用户指定返回的文件描述符的值, 如果需要, 则首先接近 newfd 的值; 它通常用来重新打开或重定向一个文件描述符。它们的原型如下:

```

#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);

```

dup 和 dup2 都返回新的描述符, 或者返回-1 并设置 errno 变量。新老描述符共享文件的偏移量 (位置)、标志和锁, 但不共享 close-on-exec 标志。程序清单 11.7 示范了如何使用 dup2 把标准输出 (文件描述符为 1) 重定向到一个文件上。函数 print_line 使用一种更安全的版本 snprintf 格式化输出消息。

程序清单 11.7 使用 dup2 重定向 stdout

```

/*
 * dup.c - Using dup2 to redirect stdout
 */
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>

```



```
void print_line(int n);

int main(void)
{
    int fd;

    /* Scribble on stdout */
    print_line(1);
    print_line(2);
    print_line(3);

    /* Redirect stdout to the file junk.out */
    if((fd = open ("junk.out", O_WRONLY | O_CREAT, 0666)) < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    if((dup2(fd, STDOUT_FILENO)) < 0) {
        perror("dup2");
        exit(EXIT_FAILURE);
    }

    /* Scribble on redirected stdout */
    print_line(4);
    print_line(5);
    print_line(6);

    close(fd);
    close(STDOUT_FILENO);

    exit(EXIT_SUCCESS);
}

void print_line(int n)
{
    char buf[80];

    snprintf(buf, sizeof(buf), "Line #%d\n", n);
    write(STDOUT_FILENO, buf, strlen(buf));
}
```

执行命令 **make dup** 编译 **dup** 程序。运行 **dup** 的结果如下：

```
$ ./dup
Line #1
Line #2
Line #3
$ cat junk.out
Line #4
Line #5
Line #6
$
```

11.3.11 使用 select 同时读写多个文件

select 调用启用了 I/O 多路转接 (multiplexing) 功能, 这个术语意味着同时从多个文件描述符读取数据或者向多个文件描述符写入数据。多路转接的例子包括 Web 浏览器, 它能打开多个网络连接, 下载一个网页中它所能下载的内容。另一个例子是客户机/服务器应用程序, 它能同时向数十乃至数百个用户提供服务。多路转接虽然从概念上易于理解, 但是却很难高效地实现。select 调用是非资源密集型的, 这意味着要等待一定数量的文件描述符来改变状态。该调用在 `<unistd.h>` 中的原型如下:

```
int select(int n, fd_set *readfds, fd_set *writefds,
           fd_set exceptfds, struct timeval *timeout);
```

select 监视如下集合的文件描述符:

- 在 readfds 中的文件描述符集合, 用于可读取字符
- 在 writefds 中的集合中查看它们能否写入数据
- 在 exceptfds 中的集合, 用于异常

自然地, 如果你只对写入多个文件感兴趣, 那么你可能对包含了预备供读取的字符的文件描述符集合不太关心。实际上, 你的程序可能没有任何读取操作。如果是这种情况, 你可以给这个参数传递 NULL。例如, 为了忽略可读取的文件描述符以及具有某些异常条件 (类似于出错) 的文件描述符, 你可以这样调用 select:

```
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
fd_set *writeable_fds;
select(maxfds, NULL, writefds, NULL, 10);
```

参数 timeout 决定了 select 将会阻塞多久, 或者说在它把控制权返回给调用它的进程之前等待多久。如果 timeout 设置为 0, select 就会立即返回。当 I/O 操作没有等待就立即返回时, 称之为非阻塞式 (non-blocking) I/O 调用。如果你想要等待直到有 I/O 操作发生 (也就是说, 直到 readfds 或 writefds 改变) 或者直到发生错误, 那么使用类似前面的例子中的语法给参数 timeout 传递 NULL 值。

第一个参数 n 包含了在任何受监视集合中最高编号的文件描述符再加 1 (示例程序显示了决定这个值的一种方法)。如果出现错误, select 返回 -1 并且设置 errno 变量为一个恰当值。在出现错误的情况下, select 也使所有的文件描述符集合和 timeout 为空, 所以在重新使用它们之前, 你必须把它们重新设置为有效值。如果 select 调用成功, 它返回的不是在受监视 (非空) 的文件描述符集合中包含的描述符总数, 就是 0。返回值为 0 意味着没有任何“有趣”的情况发生; 也就是说, 在 timeout 失效之前没有描述符改变状态。

select 实现也包含了 4 个示例程序处理描述符集合:

```
FD_ZERO(fd_set *set);
FD_SET(int fd, fd_set *set);
FD_CLR(int fd, fd_set *set);
```

```
FD_ISSET(int fd, fd_set *set);
```

它们的操作如下:

- `FD_ZERO` 清除集合 `set`。
- `FD_SET` 把描述符 `fd` 添加到 `set`。
- `FD_CLR` 从 `set` 中删除 `fd`。
- `FD_ISSET` 判断 `fd` 是否在 `set` 中。`FD_ISSET` 例程在 `select` 返回后使用以判断是否发生了需要采取行动的情况。如果 `fd` 是集合, 它的状态在 `select` 调用期间改变 (它包含要读取的字节、能写入的字节, 或者出现了错误)。

下面的例子 `mplex` 监视两个有名管道, 查看是否有数据可读取 (示例程序使用管道是因为它们是在短小的程序中展示多路转接 I/O 操作最简单的途径)。

注意: 下面的程序 `mplex.c` 取自 `mpx-select.c`, 它首次出现在 Micheal K. Johnson 和 Erik W. Troan 编写的 “Linux Application Development” 一书 (Addison Wesley, 1998) 中的 213 ~ 214 页。

除了这个程序之外, 你还需要使用如下命令创建两个有名管道 (位于和二进制文件 `mplex` 相同的路径下):

```
$ mknod pipe1 p
$ mknod pipe2 p
```

接下来启动 `mplex`。打开另外两个终端窗口。在第一个窗口中键入 `cat > pipe1`, 而在第二个窗口中键入 `cat > pipe2`。此后你在两个窗口中键入的任何内容都排队供 `mplex` 读取。程序代码如下:

```
/*
 * mplex.c - read input from pipe1 and pipe2 using select
 *
 * Adapted from mpx-select.c, written
 * by Michael Johnson and Erik Troan. Used with the permission of
 * the authors of Linux Application Development,
 * Michael Johnson and Erik Troan.
 */
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define BUFSZ 80

void err_quit(char *msg);

int main(void)
{
    int fds[2];
    char buf[BUFSZ];
```

```

int i, rc, maxfd;
fd_set watchset; /* Read this set of file descriptors */
fd_set inset; /* Copy of watchset for select to update */

/* Open the pipes */
if((fds[0] = open ("pipe1",O_RDONLY | O_NONBLOCK))< 0)
    err_quit("Open pipe1");
if((fds[1] = open ("pipe2",O_RDONLY | O_NONBLOCK))< 0)
    err_quit("open pipe2");

/* Initialize watchset with our file descriptors */
FD_ZERO(&watchset);
FD_SET(fds[0],&watchset);
FD_SET(fds[1],&watchset);

/* select needs to know the maximum file descriptor */
maxfd = fds[0] > fds[1] ? fds[0] : fds[1];

/* Loop while watching the pipes for output to read */
while(FD_ISSET(fds[0],&watchset) || FD_ISSET(fds[1],
        &watchset)) {
    /* Make sure select has a current set of descriptors */
    inset = watchset;
    if(select(maxfd + 1,&inset, NULL,NULL,NULL) < 0)
        err_quit("select");
    /* Which file descriptor is ready to read? */
    for (i = 0; i < 2;++i) {
        if(FD_ISSET(fds[i],&inset)) {
            rc = read(fds[i],buf,BUFSZ - 1);
            if(rc > 0) { /* Read some data */
                buf[rc] = '\0';
                printf("read: %s", buf);
            } else if (rc == 0) { /* This pipe is closed */
                close(fds[i]);
                FD_CLR(fds[i],&watchset);
            } else
                err_quit("read"); /* Bummer */
        }
    }
}
exit(EXIT_SUCCESS);
}

void err_quit(char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}

```

和通常一样，使用命令 `make mplex` 编译这个程序。`mplex` 稍有点复杂。在打开两个有名管道之后，它使用 `FD_ZERO` 初始化一个文件描述符集合，然后加入两个管道的描述符。`mplex` 的核心是 `while` 循环。在每次循环中，它使用 `FD_ISSET` 查看两个管道中是否有一个管道有数据可读取。如果有，则首先把 `watchset` 复制到 `inset`，因为 `watchset` 可能因为其中有一管道被关闭而改变。

接下来进行 `select` 调用，然后的 `for` 循环判定哪一个描述符做好了供读取的准备（`timeout` 的值为 `NULL`，则 `select` 阻塞直到数据准备好可供读取）。接着 `for` 循环读取管道并显示数据。如果 `read` 返回 0，管道已被关闭，于是 `mplex` 关闭那个文件并把它从 `watchset` 中删除。当最后一个管道关闭时，`mplex` 终止执行。

11.3.12 使用 `ioctl`

系统调用 `ioctl` 的作用是设置或检索文件的多种有关参数并对文件进行一些其他的操作。是否可以使用 `ioctl` 以及传递给 `ioctl` 什么参数随下层设备的不同而不同，请查看 `ioctl` 的手册页面可以了解完整的详细信息。

```
#include <sys/ioctl.h>
int ioctl(int d, int request, ...);
```

参数 `d` 必须是一个打开的文件描述符。

11.4 小 结

本章介绍了多种基于文件描述符的输入和输出函数。在讨论了诸如文件类型、权限和所有权这样的基本 Linux 文件特性后，本章接着转向详细讨论文件描述符。特别地，你学习了一系列使用文件描述符的系统调用，包括 `open`、`creat`、`close`、`read`、`write`、`ftruncate`、`lseek`、`fsync`、`fstat`、`fchown`、`fchmod`、`flock`、`fcntl`、`dup`、`dup2`、`select` 和简要介绍的 `ioctl`。下一章将考虑 C 标准库提供的 I/O 工具。

第 12 章 文件和目录操作

本章的目的是让你快速了解标准 C 的 I/O 库，并且介绍在 Linux 编程环境里如何操作目录。头文件 `<stdio.h>` 中声明了标准 C 的 I/O 库，而标准 C 的 I/O 库在所有通用计算机上（但在某些嵌入式系统上）的 C 语言实现都是相同的。大多数 C 程序员应该已经相当熟悉它了。在快速回顾这些基础知识以后，本章转向讨论如何操作目录，并以说明两个仅由 Linux 的 ext2 文件系统具有的特性作为本章的结束。

在本章中描述的函数是库函数而不是系统调用。库函数和系统调用间的区别在于系统调用能够让你直接访问 Linux 内核提供的丰富服务，比如上一章讨论的基于文件描述符的 I/O 操作。于是，你可以把系统调用看作是内核的低级接口。另一方面，库调用处于 Linux 的编程接口中较高的层次。实际上，许多库函数都是用系统调用来实现的，例如内存分配例程 `malloc` 就是由系统调用 `sbrk` 实现的。库函数和系统调用之间第二个关键区别在于系统调用存在于内核空间，而大多数库调用都是用户模式的例程。因此，系统调用，特别是被具有超级用户权限的进程调用的系统调用，可能会破坏系统。另一方面，运行库调用损害运行中的系统的风险要小得多。

标准 I/O 库对文件描述符的 I/O 操作有几点增强。它主要的优点是对 I/O 操作进行缓冲，减少了系统调用的开销。但使用缓冲的缺点是导致输出不在期望的时刻被传送出去，而且忽视了块的大小，这对于向磁带写入数据和使用某种网络协议来说很重要。幸好可以禁用缓冲。最流行的 `printf` 函数族使用文件指针 I/O，而不是文件描述符 I/O，并且还有许多其他面向行的函数。这些函数还可以让被信号中断的系统调用继续执行。库的可移植性也是一个主要优点。

12.1 标准文件函数

在随后的函数描述中，`FILE *` 标识出一个文件指针。本节描述的大多数函数都接受一个文件指针作为参数来指出对哪个流进行操作。回忆前面章节的内容，每个进程通常有 3 个文件——`stdin`、`stdout` 和 `stderr`。除非专门做改动，不是在编程上改变就是使用管道或重定向改变，否则这些文件指针指向与用户终端关联的流。

警告： 本章描述的一些函数可能实际上是由宏实现的，因此需注意使用表达式作为这些函数的参数可能会带来副作用，因为结果可能导致程序出现不可预料的行为。

12.1.1 打开和关闭文件

`fopen`、`freopen` 和 `fclose` 调用是 ANSI 标准库的一部分；`fdopen` 不是。它们的原型为：

```
#include<stdio.h>
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fildes, const char *mode);
FILE *freopen(const char *path, const char *mode, FILE *stream);
int fclose(FILE *stream);
```

`fopen` 以模式 `mode` 打开名为 `path` 的文件。表 12.1 介绍了文件的模式。注意 Linux 不区分文本模式和二进制模式——这在对行结尾做不同处理的操作系统，如 DOS、MS-Windows 和 MacOS 上很重要。`fopen` 返回一个文件指针，该指针可以传递给别的标准 I/O 函数来标识这个流。文件指针指向一个描述流的结构。在出现错误的情况下，`fopen` 返回 NULL 并把 `errno` 变量设置为恰当的值。如果用 `fopen` 打开供写入数据的文件不存在，那么它会以权限 0666 来创建该文件，这和用进程的 `umask` 来设置权限的情况类似。

表 12.1 有效的文件模式

模式	读	写	位置	截断	创建	二进制
r	Yes	No	Beginning	No	No	Text
r+	Yes	Yes	Beginning	No	No	Text
w	No	Yes	Beginning	Yes	Yes	Text
w+	Yes	Yes	Beginning	Yes	Yes	Text
a	No	Yes	End	No	Yes	Text
a+	Yes	Yes	End	No	Yes	Text
rb	Yes	No	Beginning	No	No	Binary
r+b or rb+	Yes	Yes	Beginning	No	No	Binary
wb	No	Yes	Beginning	Yes	Yes	Binary
w+b or wb+	Yes	Yes	Beginning	Yes	Yes	Binary
ad	No	Yes	End	No	Yes	Binary
a+b or ab+	Yes	Yes	End	No	Yes	Binary

`freopen` 打开在 `path` 中指定的文件，并把它和 `stream` 指向的文件关联起来。这个函数把文件指针关闭后重新打开，于是文件指针指向新的文件。对于下层的文件描述符 I/O 操作，它也应该使用相同的文件描述符。`freopen` 典型的用途在于重定向流 `stdout`、`stdin` 和 `stderr`，这和使用 `dup2` 重定向文件描述符很相似。如果发生错误，`freopen` 返回 NULL 并且设置 `errno` 变量。

`fdopen` 函数把一个文件指针和文件描述符 `fildes`（由 `open`、`pipe` 或 `accept` 调用所创建）关联起来。如果出现错误，`fdopen` 返回 NULL 并且设置 `errno` 变量。

最后为了关闭文件流，使用函数 `fclose`。`fclose` 执行成功则返回 0，而如果执行失败则返回 EOF，此时它还会设置 `errno` 变量。一旦文件由 `fclose` 关闭，任何试图对这个被关闭流的访问，包括其他 `fclose` 调用都会导致出现不可意料的结果。

12.1.2 读写文件

函数 `fread` 和 `fwrite` 允许从文件流读出数据以及向文件流写入数据。它们的原型如下：

```
#include<stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(void *pnt, size_t size, size_t nmemb, FILE
*stream);
```

指针 `ptr` 指向的缓冲区保存 `fread` 从文件读入的数据或者保存 `fwrite` 向文件写回的数据。通常由 `stream` 指定要操作的数据流。`size` 和 `nmemb` 分别控制读入或写回的一条记录的大小和记录数。但是，在这里使用“记录”这个词对于 Linux 来说有点儿不太合适，因为 Linux 并不是以记录的方式来完成读写操作的。此时 `size` 是指读写的字节数，而 `nmemb` 是指读写多少个单位的 `size`。`fread` 返回读入的记录数，`fwrite` 返回写回的记录数而不是字节数。两种函数的返回值都有可能比请求值少，所以可以使用下一节讨论的 `feof` 或 `ferror` 调用检查出错状态。在出现错误的情况下，返回值通常比请求值要小些，而且可能取负值。这个时候可以再使用 `feof` 和 `ferror` 来判断操作失败的原因。

12.1.3 获得文件状态

`feof` 和 `ferror` 函数都返回流的当前状态。`clearerr` 清除在文件上已经设置的错误位。`fileno` 返回与给定的文件流相关联的文件描述符。它们的原型如下：

```
#include <stdio.h>
int feof(FILE *stream);
int ferror(FILE *stream);
void clearerr(FILE *stream);
int fileno(FILE *stream);
```

如果遇到 EOF 则 `feof` 返回非零值。但要注意一点，通常只有在执行读操作的位置确实超出了文件末尾才设置 EOF 标志。因此，使用如下结构的循环才可能使循环超出文件末尾：

```
while(!feof(stream)) {
    /* do stuff */
}
```

于是需要用另一种方法检测 EOF。例如，如果一次读出一个字符，可以使用下面的结构：

```
int c;
while((c = fgetc(stream)) != EOF) {
    /* do stuff */
}
```

提示： 注意 `fgetc` 返回一个整数而不是一个字符。试图把 `fgetc` 的返回值保存在一个 `char` 类型的变量中是一种常见的编程错误，这会导致出现无限循环的结果。

如果在流上设置了出错标志则 `ferror` 返回一个非零值。注意这个函数不设置 `errno` 变量。在这种情况下，`errno` 变量由前一次函数调用来进行设置。调用 `clearerr` 可以清除流的 EOF 标志和出错标志。

函数 `fileno` 返回和一个与流相关联的文件描述符，供用户执行基于文件描述符的 I/O 操作使用。例如，如果你要向一个文件发出 `fstat` 系统调用，可以使用 `fileno` 检索它的文件描述符。但使用这种方法要注意查看标准 I/O 库的头文件。通常，把标准 I/O 库中的调用和其他 I/O 调用（如 `read` 或 `write`）混合使用是不明智的做法，因为这会导致出现不可预料的结果。如果必须要混用，则要保证在此之前调用 `fflush` 把缓冲区内容强制写入到物理设备上。

12.2 输入输出调用

标准的 C 语言 I/O 库有丰富的输入输出函数集。本节讨论用于格式化输出的 `printf` 函数族、用于格式化输入的 `scanf` 函数族、用于字符输入和输出的例程、基于行的输入和输出调用，以及取得和设置文件指针的调用，它们可以让你从文件的任意位置分别读出和写入数据。如前所述，最后你会学到控制标准 I/O 库的输入输出缓冲行为的函数。

12.2.1 格式化输出

稍有 C 编程经验的人都应该很熟悉 `printf` 函数族。多种 `printf` 例程都包含了许多可选参数，能够向任意的输出流打印输出格式的结果。它们的原型为

```
#include <stdio.h>
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);

#include <stdarg.h>
int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
int vsnprintf(char *str, size_t size, const char *format, va_list ap);
```

`printf` 函数有多种有趣的变体。那些以“s”开头的函数向一个字符串而不是一个流输出结果。以“s”开头的变体函数天生不健全，因为它们不检验传递给它们的参数的长度，因而很容易遭受以溢出错误为手段的攻击。取而代之的是以“sn”开头的函数；它们不会遭受缓冲溢出的问题。但缺点是在许多其他操作系统上没有以“sn”开头的函数。因此，如果你需要把自己的程序移植到没有修正缓冲溢出问题的操作系统上，那么即使不检查缓冲区溢出的问题，也要用一个 `vsprintf` 的包裹函数来实现 `snprintf` 的调用规则。在不健全的系统上你的代码也易被攻破，但在修正了安全问题的系统上则工作正常。

`printf` 函数族中以“v”开头的函数非常灵活。它们是带有不定参数（`vararg`）的版本，你用它们编写的函数可以处理数目、类型不确定的参数如和 `printf` 函数族一样的情形。

在 `<stdarg.h>` 中声明的宏 `vararg` 解决了传递给函数的参数个数以及类型都不确定的问题。这种函数必须至少有一个固定参数。函数原型的参数列表必须包含“...”以表明不定

参数从这里开始。它告诉编译器，当你向这个函数传递额外的而且类型似乎不一致的参数时不要报错。

注意： 省略号还告诉编译器使用对 `stdarg` 友好的调用规则，如把参数保存在寄存器中以便更快速地进行访问。例如，RISC 处理器是在寄存器而不是在堆栈中传递函数的部分或全部参数的。

`stdarg` 还提供了把一个函数的不定参数传递给另一个函数的途径，这种途径安全而且具有可移植性。这种功能最常见的用法之一就是编写自己的 `printf` 风格的函数，而这些函数反过来又使用已有的函数来实现。

12.2.2 格式化输入

`fscanf` 函数用于从 `stdin` 读取控制字符串 `format` 规定的格式化输入。`fscanf` 有许多变体函数可以从任意流（它们的名字以“f”开头）或者一个字符串（它们的名字以“s”开头）读取数据。名字以“v”开头的变体函数使用了宏 `stdarg`。这些变体函数和 `printf` 函数族类似。

```
#include <stdio.h>
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);

#include <stdarg.h>
int vscanf(const char *format, va_list ap);
int vsscanf(const char *str, const char *format, va_list ap);
int vfscanf(FILE *stream, const char *format, va_list ap);
```

人们使用这些函数时最常见的错误之一是给函数传递了变量的值而不是变量的地址（指针）；别忘了在必要的地方加上“&”操作符。通常，字符串不需要加“&”操作符而其他变量却需要加。

在控制字符串中没有提供字符串参数的长度会引起缓冲区溢出，从而产生严重的安全后果。如果你提供了长度，那么在特定输入域上任何多余的输入都可能会导致输入终止在出错的输入域上。

注意，任何直接从流读取数据的 `scanf` 版本有可能出现这样的问题，在一行中多余的数据会被后续的 `scanf` 函数读入。因此，最好先把一行数据读入到字符串缓冲中，再用 `sscanf` 处理字符串缓冲。把数据读入到一个字符串缓冲的做法还可以让你对同一个字符串使用多个 `scanf` 调用，但它们具有不同的格式，以便处理多种可能的输入格式。

`scanf` 函数成功读入数据后返回输入域的个数，如果出错则返回 EOF 或者比预期输入域个数小的值。

12.2.3 字符输入输出

使用下面的函数可以一次读入或写出一个字符：

```
#include <stdio.h>
```

```

int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);

```

注意，`getchar` 和 `putchar` 通常是由宏实现的，所以在有的地方使用它们要小心，因为副作用可能产生意外的结果。

12.2.4 行输入输出

有许多函数支持基于行的输入输出。它们的原型为

```

#include <stdio.h>
char *fgets(char *s, int size, FILE *stream);
char *gets(char *s);
int fputs(const char *s, FILE *stream);
int puts(const char *s);

```

表 12.2 对这些函数做了简要总结。返回值的类型为整数的函数在出错时返回 EOF 并且设置 `errno` 变量。返回值的类型为字符指针的函数在正常情况下返回指向读出字符串的指针，出错时返回 NULL。

表 12.2 比较字符和行 I/O 函数

函数	方向	尺寸	流	溢出	新一行
<code>fgetc</code>	Input	Character	Any	No	
<code>fgets</code>	Input	line	Any	No	Kept
<code>getc</code>	Input	Character	Any	No	
<code>getchar</code>	Input	Character	stdin	No	
<code>gets</code>	Input	Line	stdin	Yes	Removed
<code>ungetc</code>	Input	Character	Any	No	
<code>fputc</code>	Output	Character	Any	No	
<code>fputs</code>	Output	Line	Any	No	Not added
<code>putc</code>	Output	Character	Any	No	
<code>putchar</code>	Output	Character	stdout	No	
<code>puts</code>	Output	Line	stdout	No	Added

面向行的输入函数对于新行的处理方式是不一致的。函数 `gets` 不能很好地处理缓冲区溢出的问题，应尽量少用这个函数；可以使用 `fgets` 来代替它，但要注意它们处理新行的方式不同，`fgets` 保留新行的行终止符，而 `gets` 却不保留。

12.2.5 文件定位

文件定位函数设置文件内部的当前位置；它们对没有指向普通文件的流，如套接口和管道不起作用。这些函数的原型如下：

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, fpos_t *pos);
void rewind(FILE *stream);
```

`fseek` 函数把当前位置设定到 `offset` 处。参数 `whence` 可以是 `SEEK_SET`、`SEEK_CUR` 或 `SEEK_END`；这些值决定了是相对于文件的起始，还是文件的当前位置或者文件的末尾来计算偏移量 `offset`。正常情况下，`fseek` 返回文件指针相对于文件起始位置的偏移量，如果出错则返回 -1（并且检查 `errno`）。`ftell` 函数只是简单地返回当前位置。`rewind` 函数把文件指针设置为 0；也就是说，把文件指针设置到文件的起始位置。`fgetpos` 和 `fsetpos` 函数是 `ftell` 和 `fseek` 函数的变体实现；在其他一些不把文件作为简单数据字节流的操作系统上，`fpos_t` 可能是个结构而非整数。

12.2.6 缓冲区控制

下面列出了缓冲区控制函数的原型。这些函数提供了 3 种主要的流缓冲机制：无缓冲、行缓冲和块缓冲，另外它们提供了将缓冲区中尚未写入设备的数据强制写入设备的功能。

```
#include <stdio.h>
int fflush(FILE *stream)
int setbuf(FILE *stream, char *buf)
int setbuffer(FILE *stream, char *buf, size_t size);
int setlinebuf(FILE *stream);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

函数 `fflush` 把缓冲区中的尚未写入设备的数据强制写入到输出流 `stream` 上。`setvbuf` 例程设置流使用的缓冲区。其参数为指向流的文件指针 `stream`、流所用到的缓冲区 `buf`、文件模式 `mode` 以及缓冲的大小。文件模式 `mode` 的取值可以为 `_IONBF`（用于无缓冲操作）、`_IOLBF`（用于行缓冲）和 `_IOFBF`（用于完全缓冲）。要改变流的缓冲模式，只需简单地调用缓冲地址为 `NULL` 的 `setvbuf` 函数即可——此时缓冲不受影响而缓冲模式会被改变。其他函数基本上都是 `setvbuf` 函数功能更多样化的变体。注意：只能在打开一个流或者调用 `fflush` 函数之后再使用 `setvbuf` 函数。如果在缓冲区中还有数据的时候调用它，会失去缓冲区中的数据。

提示： 代码片段 `setbuf(stream, NULL)`；经常用来代替函数 `setvbuf` 取消对流的缓冲。

12.2.7 删除和改名

函数 `remove` 根据提供的文件名删除文件，而函数 `rename` 能够改变一个文件的名称。

```
#include <stdio.h>
int remove(const char *pathname);
int rename(const char *oldpath, const char *newpath);
```

两个函数的第一个参数都是一个现有文件的路径名。函数 `rename` 的第二个参数是文件要修改成的新路径名。两个函数在执行成功时都返回 0，在执行失败时都返回 -1。通常，如果它们执行出错还会把特定的错误代码保存在变量 `errno` 中。

12.2.8 使用临时文件

函数 `tmpfile` 和 `tmpnam` 是 ANSI 标准 C 库的组成部分；另外两个 (`mkstemp` 和 `mktemp`) 则是 UNIX 系统所特有的函数。

```
#include <stdio.h>
FILE *tmpfile(void);
char *tmpnam(char *s);

#include <unistd.h>
int mkstemp(char *template);
char *mktemp(char *template);
```

函数 `tmpfile` 打开一个临时文件，返回一个指向 `FILE` 结构的指针。函数 `tmpnam` 用于产生临时文件的文件名。如果字符串 `s` 不为 `NULL`，则文件名被写入所提供的缓冲区中（由于字符串参数没有限制大小，所以有可能溢出）。否则，函数 `tmpnam` 返回一个指向内部缓冲区的指针，下次调用 `tmpnam` 时会重写这个缓冲区。不幸的是，这两个函数都不能让你指定保存文件的位置，它们可能会在诸如 `/tmp` 或 `/var/tmp` 这样脆弱的共享目录下创建临时文件（或路径名）。因此，除非你无法使用其他调用（`mkstemp` 或 `mktemp`），否则不应该使用这两个函数。

下面的代码片段给出了一个如何使用 `tmpnam` 创建并打开一个临时文件的范例：

```
FILE *fp;
char somefile[256];
f = tmpnam(NULL);
if ((fp = fopen(f)) != NULL) {
    /*
     * do stuff here
     */
}
fclose(fp);
```

函数 `mktemp` 也能用于创建惟一的临时文件名，但是它使用了模板，可以为文件名指定其路径前缀；模板的最后 6 个字符必须是“XXXXXX”。函数 `mkstemp` 先调用 `mktemp` 来产生一个文件名，然后发出系统调用 `open`，打开它以便进行对文件描述符的 I/O 操作。你可以用函数 `fdopen` 在这个文件描述符上打开一个标准 I/O 流。

临时文件应该只在安全的目录下创建，其他人不能对此目录有写权限（如 `~/tmp` 或 `/tmp/$`（用户名））；否则它们在争用条件下会显得很脆弱。这些函数产生的文件名也容易被猜中。

12.3 目录操作

目录完全符合 Linux 的文件抽象概念，所以称之为目录文件可能会更恰当些，因为它们只是包含了目录下保存的文件名列表的简单文件而已。然而，处理目录需要使用特殊的编程接口。这个特殊接口让程序能够获得并使用和目录相关联的附加信息。

12.3.1 找到当前目录

用于找到当前工作目录的调用叫做 `getcwd`，它在 `<unistd.h>` 中进行声明。它的原型如下：

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

函数 `getcwd` 把当前工作目录的绝对路径名复制到 `buf` 中，该缓冲有 `size` 个字节长。如果 `buf` 不够大，不能装下整个路径名，则 `getcwd` 会返回 `NULL` 并且把 `errno` 变量的值设为 `ERANGE`。如果出现了这种情况，可增加 `buf` 的大小再试试。另一种方法为，如果 `buf` 为 `NULL` 而 `size` 又小于 0，`getcwd` 会使用 `malloc` 动态地为 `buf` 分配足够的内存。如果你利用了这一扩展特性，必须记住释放缓冲以避免内存泄漏。

12.3.2 改变目录

函数 `chdir` 或 `fchdir` 都能改变当前目录，它们的原型为

```
#include <unistd.h>
int chdir(const char *path);
int fchdir(int fd);
```

函数 `chdir` 把当前目录改为 `path` 所包含的新目录。函数 `fchdir` 功能类似，只是必须传递给它一个打开的文件描述符 `fd`。

12.3.3 创建和删除目录

幸运的是，用于创建和删除目录的函数名和它们在命令行的对应命令名 `mkdir` 和 `rmdir` 相同。使用 `mkdir` 需包含 `<fcntl.h>` 和 `<unistd.h>` 两个头文件；而使用 `rmdir` 只需要 `<unistd.h>` 头文件。它们的原型为

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
```

函数 `mkdir` 会尝试以 `mode` 为权限建立 `pathname` 指定的目录，这和 `umask` 修改文件的权限类似。函数 `rmdir` 删除 `pathname` 指定的目录，这个目录必须为空。两个函数在执行成功后返回 0，执行失败则返回 -1 并设置 `errno` 变量。

12.3.4 获得目录列表

列出目录内容意味着读出目录文件的内容。基本处理操作并不复杂：

1. 使用 `opendir` 函数打开目录文件。
2. 使用 `readdir` 函数读出目录文件的内容，如果你已经读到了目录文件的末尾，还想再从头开始，则可以使用 `rewinddir` 函数把文件指针重定位到目录文件的起始位置。
3. 使用 `closedir` 函数关闭目录文件。

所有这些函数都在头文件 `<dirent.h>` 中声明。它们的原型为

```
#include <dirent.h>
#include <sys/types.h>
DIR *opendir(const char *pathname);
struct dirent *readdir(DIR *dir);
int rewinddir(DIR *dir);
int closedir(DIR *dir);
```

`opendir` 函数打开 `pathname` 所指定的目录（它毕竟是一个文件），返回一个指向 `DIR` 流的指针。否则，如果出错，该函数返回 `NULL`，并且设置 `errno` 变量。流指针定位于流的起始位置。显然，`closedir` 函数用于关闭流 `dir`，如果执行成功则返回 0，否则返回 -1。`rewinddir` 函数把流指针移回到流的起始位置。它也在执行成功后返回 0，执行失败后返回 -1。

`readdir` 函数执行了读取目录内容的大部分工作。它返回一个指向 `dirent` 结构的指针，这个结构包含了来自 `dir` 的下一条目录内容项。以后每次调用 `readdir` 都用新数据覆盖返回的 `dirent` 结构。直到达到文件末尾或者出错时，`readdir` 才返回 `NULL`。要取得文件名，可以使用 `dirent` 结构的成员 `dirent.d_name[]`，它返回指向文件名的指针。

警告： `dirent` 结构只有一个成员是可移植的（也就是说，由 POSIX 定义并且在所有兼容 POSIX 的系统上都能够预料其行为）：`d_name[]`。其他所有成员都由系统定义，并与系统相关，所以它们的名称以及包含的数据类型可能随系统的不同而不同。例如，某些系统还限制文件名最多为 14 个字符，而其他像 Linux 等系统允许长达 256 个字符的文件名。

12.4 特殊的 ext2 文件系统属性

能用 ext2 文件系统在文件上设置多达 4 种特殊的属性：

- 固定不变的（Immutable）——`EXT2_IMMUTABLE_FL`
- 只能添加的（Append-only）——`EXT2_APPEND_FL`
- 不能卸出的（No-dump）——`EXT2_NODUMP_FL`
- 同步（Sync）——`EXT2_SYNC_FL`

只有超级用户才可以设置或清除 Immutable 和 Append-only 标志,但是文件属主可以设置或清除 No-dump 和 Sync 标志。这些标志的含义是什么呢?下面详细列出了对它们的介绍:

- 固定不变的 (Immutable) 文件根本不能修改: 你不能向其中加入数据, 不能删除或改变它们的名字, 也不能增加到它们的链接。甚至超级用户也不能执行上述操作——在操作之前必须首先清除 Immutable 标志。
- 只能添加的 (Append-only) 文件只能以添加模式写入数据, 而且也不能删除、改名或被链接。
- 不能卸出的 (No-dump) 属性影响通常用于创建备份的 dump 命令, 让该命令忽略某个文件。
- 同步 (Sync) 属性影响同步写入的文件; 也就是说, 所有向该文件执行写入操作的 write 调用必须在返回之前完成写入操作 (其作用等价于调用带有 O_SYNC 选项的 open 调用)。

为什么要使用这些属性? 让文件成为“固定不变的 (Immutable)”, 能够避免该文件被意外删除或修改, 所以这是保护重要文件的一种方便的安全手段。“只能添加的 (Append-only)”标志在保留文件当前内容的同时还允许你向其中增添数据——这同样也是一种方便的安全预防措施。

“不能卸出的 (No-dump)”标志只是在你备份系统时节省宝贵的空间和时间的一种便捷手段。最后, “同步 (sync)”标志尤其有助于保证关键文件, 比如数据库, 能够按要求确实写入数据。如果系统在缓冲数据还没以物理方式写入到硬盘之前就崩溃, 那么使用“同步 (sync)”标志可以避免丢失数据。但是采用“同步 (sync)”标志会极大地降低程序的性能。

要取得或设置这些属性, 可以使用 ioctl 调用, 它在 <sys/ioctl.h> 中声明。其原型如下:

```
int ioctl(int fd, int request, void *arg);
```

上述标志的声明位于 <linux/ext2_fs.h>。要检索文件描述符 fd 指定的文件的属性, 必须把 request 设定为 EXT2_IOC_GETFLAGS。要设置上述属性, 必须把 request 设定为 EXT2_IOC_SETFLAGS。在这两种情况下, arg 都保存了被操控的标志。

警告: 本节的内容专门针对 Linux 的主要文件系统 ext2, 它正式的称呼叫做 Second Extended 文件系统。其他版本的 UNIX 或许有我们讨论的函数和结构, 但它们肯定不会具有这里介绍的功能。如果你要在一个准备移植的程序中使用这些函数, 必须用预处理宏 #ifdef 把它们包括起来, 好让你的程序能够在非 Linux 的系统上正确编译和运行。

下面的示例程序 setext2 在作为惟一参数传递给它的文件上设置“同步 (sync)”和“不能卸出 (no-dump)”属性。它能很容易地扩充成为能够在任意文件组上设置任何 ext2 扩展属性的程序。


```
/*
 * setext2.c - Set ext2 special flags
 */
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <linux/ext2_fs.h>
#include <sys/ioctl.h>
#include <errno.h>

int main(int argc, char *argv[])
{
    int fd;
    long flags;

    /* Usage nag */
    if (argc != 2) {
        puts("USAGE:setext2 {filename}");
        exit(EXIT_FAILURE);
    }

    if((fd = open(argv[1], O_RDONLY)) < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    /* These are the flags we'll set on the file */
    flags = EXT2_SYNC_FL | EXT2_NODUMP_FL;
    if(ioctl(fd, EXT2_IOC_SETFLAGS,&flags)) {
        perror("ioctl");
        close(fd);
        exit(EXIT_FAILURE);
    }

    if (flags & EXT2_SYNC_FL)
        puts("SYNC flag set");
    if (flags & EXT2_NODUMP_FL)
        puts("NODUMP flag set");

    close(fd);
    exit(EXIT_SUCCESS);
}
```

运行 setext2 的输出结果如下:

```
$ touch foo
$ lsaddt foo
----- foo
$ ./setext2 foo
```

```
SYNC flag set
NODUMP flag set
$ lsattr foo
---S--d- foo
```

ioctl 例程要求被修改属性的文件必须已经打开，因此要使用 open 调用。在把属性值 EXT2_SYNC_FL 和 EXT2_NODUMP_FL 赋给变量 flags 后，ioctl 调用试图设置它们，如果调用失败要记得关闭文件。最后一段代码确认要求的属性（sync 和 no-dump）实际上已经设置好了。在示范运行过程中的 lsattr 命令进一步确认了这一事实。

提示： Linux 有两条命令 chattr 和 lsattr 分别用于设置和查询本节介绍的 ext2 特有的属性。简言之，chattr 用于设置特殊属性，而 lsattr 显示设置的特殊 ext2 属性。在上面示范运行的过程中，文件 foo 的属性中大写“S”说明设置了 EXT2_SYNC_FL(sync)属性，而小写“d”说明设置了 EXT2_NODUMP_FL(no-dump)属性。参考 man chattr 和 man lsattr 了解更多的细节。

注意： 好奇的读者熟读手册页面后会找到检索或操作 ext2 文件系统的 inode(i 节点)和 superblock(超级块)的系统调用。这些例程在用户级程序中很少会用到，所以本书没有讨论它们。为什么呢？简单地说，如果误用或者不正确地调用了它们，即使在最好的情况下也能破坏文件或目录。在最坏的情况下，弄乱 superblock(超级块)属性可以轻易地使整个文件系统损害，系统无法自举启动。

12.5 小 结

标准 I/O 库 (stdio 库) 中包含的基于文件指针的函数是标准 C 库的一部分，它们提供了更易于使用和移植的文件接口，特别是对文本文件。然而，在某些场合下，比如创建和操作目录时，如果你使用本章讨论的 Linux 特有的函数调用将使处理问题更容易。

第3部分 进程和同步

第13章 进程控制

本章介绍进程控制的基础知识。在较为详细地阐述 Linux 的进程模型之后，本章讨论进程的属性、包括著名的进程标识号（process identifier, PID）、进程真实的（real）和有效的（effective）ID 以及与 setUID 和 setGID 程序相关联的风险和好处。接下来，本章介绍如何使用标准 C 库提供的 system 函数，以及使用包括 fork、多种 exec 例程和 popen 在内的 Linux 特有调用来创建进程。随后一节讲述进程控制、如何等待进程、在进程上设置超时和计时器以及如何杀死进程。在专门用一节内容介绍完信号之后，本章以介绍 Linux 进程调度作为结束。

13.1 Linux 进程模型

在传统的 UNIX 模型里有两种创建或修改进程的操作。函数 fork 用于创建一个新的进程，该进程几乎是当前进程的一个完全拷贝。调用函数 execve 可以在进程中用另外的程序来替换当前运行的进程。运行另外一个程序通常包含了上述两种操作，可能还会改变前后的运行环境。

除了传统的 Linux 进程模型，另外还有两种创建进程的方法。轻量级进程，也称为线程，提供了独立的执行线索和堆栈段，但却共享数据段。Linux 特有的 __clone 调用用于支持线程；它通过指定共享的属性带来了更好的灵活性。第 14 章将向你介绍 pthread（POSIX 线程）库并简要讨论系统调用 clone。

13.2 进程属性

准确地说，什么是进程呢？一个进程是一个正在执行的程序的实例，它也是 Linux 基本的调度单位。对于一个正在运行的程序来说，一个进程由如下元素组成：

- 程序的当前上下文（context），它是程序当前执行的状态
- 程序的当前执行目录
- 程序访问的文件和目录
- 程序的信任状态（credentials）或者说访问权限，比如它的文件模式和所有权
- 内存和其他分配给进程的系统资源

内核使用进程来控制对 CPU 和其他系统资源的访问，并且使用进程来决定在 CPU 上运行哪个程序、运行多久以及采用什么特性运行它。内核的调度器负责在所有的进程间分配 CPU 执行时间，称为时间片（time slice），它轮流在每个进程分得的时间片用完后从进程那里抢回控制权。时间片非常小，小到让单处理器系统上的几个进程仿佛是在同时运行一样。每个进程还包含了有关它们自身的充分信息，必要时内核能在执行与不执行它之间进行切换。

进程也具有许多能惟一定义它们的属性和特性。进程的属性或特性能够把它们标识出来并且规定它们的行为。内核内部还维护了关于每个进程的大量信息，并且对外提供一个访问这些信息的接口。下一节讨论这些信息的内容和读取并操作信息的接口。

13.2.1 进程标识号

进程最知名的属性就是它的进程号（process ID, PID）和它的父进程号（parent process ID, PPID）。PID 和 PPID 都是非零正整数。一个 PID 惟一地标识一个进程。一个进程创建一个新进程称为创建了子进程（child process）。相反地，创建子进程的进程称为父进程（parent process）。

所有的进程追溯其祖先最终都会落到进程号为 1 的进程身上，这个进程叫做 init 进程。init 进程是内核自举后第一个启动的进程。init 引导系统、启动守护进程并且运行必要的程序。虽然系统启动过程的内幕已经超出了本书的范围，但是指出 init 是所有进程的父进程这一点是很重要的。

为什么一个进程要知道它的 PID 以及它的父进程的 PID 呢？PID 的常见用法之一就是创建惟一的文件名或目录。例如，当调用 getpid 后，进程接着要用 PID 创建一个临时文件。另一种典型的用途是把 PID 写入日志文件作为日志消息的一部分，以清楚说明是哪个进程记录下的日志消息。一个进程能够用它的 PPID 向它的父进程发送信号或其他消息。

程序清单 13.1 给出的程序能够打印它的 PID 和 PPID。

程序清单 13.1 打印 PID 和 PPID

```
/*
 * prpids.c - Print PID and PPID
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(void)
{
    printf("PID = %d\n", getpid());
    printf("PPID = %d\n", getppid());
    exit(EXIT_SUCCESS);
}
```

用命令 `make prpids` 编译这个程序。这个程序的输出应该和下面的结果类似：

```

$./prpids
PID = 21548
PPID = 21367

```

自然，在你的系统上显示出的值应该与此有所不同。

13.2.2 Real 和 Effective 标识号

除了进程的 PID 和 PPID 之外，每个进程还有其他几个标识属性，表 13.1 列出它们的名字及其 C 语言类型和返回它们的函数名。为了使用表 13.1 中第三列的函数，必须在代码中包含<sys/types.h>和<unistd.h>两个头文件。

表 13.1 进程属性

属性	类型	函数
进程 ID	pid_t	getpid(void);
父进程 ID	pid_t	getppid(void);
真实用户 ID	uid_t	getuid(void);
有效用户 ID	uid_t	geteuid(void);
真实用户组 ID	gid_t	getgid(void);
有效用户组 ID	gid_t	getegid(void);

每个进程有三个用户 ID (UID) 和三个用户组 ID (GID)。它们主要用于安全目的，如为文件赋予访问权限以及限制用户只能运行某种程序。真实用户 ID 和真实用户组 ID 代表用户真实的身份。当用户登录时从/etc/passwd 文件中读取它们。它们是你的登录名和主用户组成员身份的数字化表示。

例如，在我的系统上，UID 为 500，它对应于我的登录名 kwall。GID 为 100，它对应于 users 用户组。getuid 和 geteuid 函数分别返回调用进程的真实 ID 和有效 ID。类似地，getgid 和 getegid 函数返回调用进程的真实 GID 和有效 GID。有效用户 ID 和有效用户组 ID 主要用于安全目的，但是在大多数情况下它们和真实用户和用户组 ID 相同。真实 ID 和有效 ID 的区别主要和设置 setUID 或 setGID 的程序有关系，这是下一节讨论的话题。

13.2.3 SetUID 和 SetGID 程序

进程的真实 ID 和有效 ID 不相同的情况是正在运行中的程序设置了 setUID 或 setGID 后才出现的。之所以称为 setUID 和 setGID 程序是因为其有效 UID 或 GID 被设置为文件的 UID 或 GID 而不是执行该程序的所有者或用户组的 UID 或 GID。setUID 和 setGID 程序的目的就是让用户能够执行具有特殊权限的程序。

例如，考虑那个用来改变口令的程序 passwd。大多数 Linux 系统都把口令保存在/etc/passwd 文件中。所有用户对这个文件都有读权，而仅有超级用户 (root) 才有写权。当运行命令 ls -l /etc/passwd 后就可以清楚地看到这一点，该命令的运行结果如下：

```

$ls -l /etc/passwd
-rw--r--r--  1 root  root   891   Jun 15:29 /etc/passwd

```

结果, 为了修改这个文件, `passwd` 程序必须有超级用户权限才行, 因为只有超级用户才有对这个文件的写权。但是, 因为任何用户都能执行 `passwd`, 所以正常情况下它不能修改 `/etc/passwd` 文件。解决这个问题的办法是, `passwd` 程序是 `setUID` 为超级用户的程序。也就是说, 当执行该程序时, 它的有效 UID 被设置为超级用户的 UID, 从而让它能够修改 `/etc/passwd` 文件。快速地运行一下命令 `ls -l /usr/bin/passwd`, 就可以确认二进制可执行程序 `passwd` 的 `setUID` 为超级用户:

```
$ ls -l /usr/bin/passwd
-r-sr-xr-x 1 root root 8735 Feb 17 09:39 /usr/bin/passwd
```

在用户可执行权限位上的 `s` 表明 `passwd` 将作为一个 `setUID` 为超级用户的程序来执行。类似地, 在组可执行权限位上的 `s` 意味着程序将作为一个 `setGID` 程序运行。

注意: 实际上, 大多数现代 Linux 系统使用隐藏保密字文件隐藏口令, 所以实际的口令保存在隐藏保密字文件 `/etc/shadow` 中, 它只能由超级用户 (或具有超级用户权限的用户) 读写。

警告: `setUID` 或 `setGID` 为超级用户的程序具有严重的安全风险, 因为即使它们只由普通用户来执行, 也会拥有超级用户的权利, 所以能够访问整个系统。这样的程序能够破坏一切。当执行或创建 `setUID` 或 `setGID` 为超级用户的程序时要极为小心。

程序清单 13.2 中的程序演示了如何检索 ID。

程序清单 13.2 取得真实和有效的 ID

```
/*
 * ids.c - Print real and effective UIDs and GIDs
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(void)
{
    printf("Real user ID: %d\n", getuid());
    printf("Effective user ID: %d\n", geteuid());
    printf("Real group ID: %d\n", getgid());
    printf("Effective group ID: %d\n", getegid());
    exit(EXIT_SUCCESS);
}
```

使用 `make ids` 编译这个程序。运行该程序产生的输出和下面的结果类似:

```
Real user ID: 500
Effective user ID: 500
Real group ID: 100
Effective group ID: 100
```

改变程序使其 setGID 为超级用户后再次运行的结果如下：

```
$ su
Password:
#chmod g+s ids
#exit
$ ls -l ids
-rwxr-sr-x  1  root  root    5375 Jul 27 23:19 ids
$ ./ids
Real user ID: 500
Effective user ID: 500
Real group ID: 100
Effective group ID: 0
```

正如你所看到的那样，这个程序执行时具有了超级用户的 GID（注意，你必须以超级用户身份设置程序的 setUID 或 setGID 位）。但是，为了得到这个结果，在改变了 setGID 位后，必须以普通用户身份运行该程序。当然，在你的系统上，GID 和 UID 的值可能有所不同。

13.2.4 用户和用户组信息

虽然计算机用数字工作得挺好，但普通人甚至程序员还是感到使用名字更舒服些。幸运的是，有两种方法能把 UID 转换为人们可读的名字。getlogin 函数返回执行程序的用户的登录名。一旦你拥有了登录名，就可以把它作为参数传递给 getpwnam 函数，这个函数能够返回/etc/passwd 文件中与该登录名相应的一行完整信息。另一种方法是把进程的 UID 传递给 getpwuid 函数，这个函数也能返回/etc/passwd 文件中恰当的条目。

getlogin 的原型如下：

```
#include <unistd.h>
char *getlogin(void);
```

它返回一个指向字符串的指针，这个字符串包含有运行该进程的用户的登录名，如果没有得到这一信息，则函数返回 NULL。一旦你拥有了登录名，就可以调用 getpwnam 检索相应于用户名的 UID。它的原型为

```
#include <pwd.h>
struct passwd *getpwnam(const char *name);
```

name 必须是一个指向包含有感兴趣的用户名的字符串指针。getpwnam 返回一个指向 passwd 结构的指针。返回的 passwd 结构指针指向静态分配的内存，下次调用 getpwnam 时会覆盖其中的内容，所以如果你以后还需要这些信息，在下次调用 getpwnam 之前应该保存好 passwd 结构中的信息。程序清单 13.3 中的程序用 getlogin 和 getpwnam 显示出了是谁正在运行进程。

程序清单 13.3 getname.c

```
/*
 * getname.c - Get login names
```

```
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <pwd.h>

int main(void)
{
    char *login;
    struct passwd *pentry;

    /* Get the login name */
    if((login = getlogin()) == NULL) { /* oops */
        perror("getlogin");
        exit(EXIT_FAILURE);
    }

    /* Get the password entry for login */
    if((pentry = getpwnam(login)) == NULL) {
        perror("getpwnam");
        exit(EXIT_FAILURE);
    }

    /* Display the password entry */
    printf("user name: %s\n", pentry->pw_name);
    printf("UID      : %d\n", pentry->pw_uid);
    printf("GID      : %d\n", pentry->pw_gid);
    printf("gecos     : %s\n", pentry->pw_gecos);
    printf("home dir  : %s\n", pentry->pw_dir);

    exit(EXIT_SUCCESS);
}
```

聪明的读者会注意到程序中故意不让代码显示出口令域的内容。执行 `make getname` 编译 `getname`。该程序的输出应该和下面类似：

```
$ ./getname
user name: kwall
UID      : 500
GID      : 100
gecos     : Kurt Wall
home dir  : /home/kwall
```

`getpwnam` 函数的手册页面列出了完整的 `passwd` 结构。正如你所看到的那样，一旦你有了调用进程的 UID，就能轻而易举地获得用户的信息。

13.2.5 附加的进程信息

除了进程、用户和组 ID 之外，系统调用和库函数还有能够检索进程的其他属性，比如

资源利用情况和执行次数。注意，我说的是执行次数，而不是执行时间。这是因为 Linux 内核维护着进程的三个独立的时间值，它们如下：

- Wall clock time（墙上时钟时间）是流逝的时间
- User CPU time（用户 CPU 时间）是进程花在执行用户模式（非内核模式）代码上的时间总量
- System CPU time（系统 CPU 时间）是花在执行内核代码上的时间总量

通过调用 `times` 或 `getrusage` 可以获得这些信息。进程的资源利用情况，比如它的内存占用量只能从 `getrusage` 调用获得。在本节里，你将首先接触如何获得时间信息的内容，然后再考虑如何获得资源利用情况。

提示： `times` 或 `getrusage`，你应该使用哪一个呢？至少从理论上讲，`getrusage` 向程序员提供了更完善的进程资源利用情况的信息。说理论上讲是因为 Linux（版本 2.2.14 尚如此）只实现了 `rusage` 结构定义的 16 种资源中的 5 种。另一方面，`times` 返回的计时信息比 `getrusage` 返回的更细致。如果你只需要计时信息或者想遵循 POSIX 标准，那就采用 `times`。如果你不关心 POSIX 的兼容性或者如果你需要 `getrusage` 提供的更多信息，就转而使用它。

进程计时

`times` 函数的原型如下：

```
#include <sys/times.h>
clock_t times(struct tms *buf);
```

`times` 返回自系统自举后经过的时钟滴答数，也称为墙上时钟时间。`buf` 是指向 `tms` 结构的指针，这个结构保存了当前进程的时间。

程序清单 13.4 中的程序使用 `system` 函数执行一条外部命令。然后，它使用 `times` 调用打印输出计时信息的结果。

程序清单 13.4 resusg1.c

```
/*
 * resusg1.c - Get process times
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/times.h>
#include <time.h>
#include <unistd.h>

void doit(char *, clock_t);

int main(void)
{
    clock_t start, end;
    struct tms t_start, t_end;
```

```

    start = times(&t_start);
    /* redirect output to prevent screen clutter */
    system("grep the /usr/doc/*/* > /dev/null 2> /dev/null");
    end = times(&t_end);

    doit("elapsed", end - start);

    puts("parent times");
    doit("\tuser CPU", t_end.tms_utime);
    doit("\tsys CPU", t_end.tms_stime);

    puts("child times");
    doit("\tuser CPU", t_end.tms_cutime);
    doit("\tsys CPU", t_end.tms_cstime);

    exit(EXIT_SUCCESS);
}

void doit(char *str, clock_t time)
{
    /* Get clock ticks/second */
    long tps = sysconf(_SC_CLK_TCK);

    printf("%s: %6.2f secs\n", str, (float)time/tps);
}

```

执行 `make resusg1` 编译该程序。其输出应该类似下面：

```

$ ./resusg1
elapsed: 21.37secs
parent time  user CPU: 0.01 secssys CPU: 0.00 secs
child time   user CPU: 1.25 secssys CPU: 1.16 secs

```

通常，你的计时信息会和这里列出的有所不同。要注意的第一点是父进程积累了非常少的时间——一秒的百分之一。虽然在 13.3 节中会更详细地讨论，但这里还要说明，当程序调用 `system` 函数时，它先产生一个子进程，然后是子进程而不是父进程完成所有工作并消耗了 CPU 时间。

值得注意的第二点是进程的执行时间 21.37 秒并不等于用户 CPU 时间和系统 CPU 时间之和，即 2.42 秒。这种明显的差异是因为子进程执行的 `grep` 操作是 I/O 密集型而非 CPU 密集型的操作。它扫描了这里讨论所使用的系统上 2331 个头文件，这些文件大约是 10MB 的文本。缺少的 18.95 秒全部用于从硬盘读取数据。

`times` 的返回值是相对而非绝对时间（系统自举后经过的时钟滴答数），所以要让它有实用价值，就必须做两次测量并使用它们的差值。这就引入了流逝时间，或者称为墙上时钟时间。`resusg1` 通过把起止的时钟滴答数分别保存在 `start` 和 `end` 中来做到这一点。另一种进程计时值可以从 `<sys/times.h>` 中定义的 `tms` 结构中获得。`tms` 结构保存着一个进程及其子进程使用的当前 CPU 时间。它的定义如下：

```

struct tms{
    clock_t tms_utime;      /* User CPU time */

```

```

    clock_t tms_stime;      /* System CPU time */
    clock_t tms_cutime;     /* User CPU time of children */
    clock_t tms_cstime;     /* System CPU time of children */
};

```

这些值也都是时钟滴答数而不是秒数。使用 `sysconf` 函数能够把时钟滴答数转换为秒数，这个函数把它的参数转换成在运行时定义的系统限制值或选项值。`_SC_CLK_TCK` 是定义每秒钟有多少滴答的宏；`sysconf` 返回值的类型为 `long`，程序用它来计算进程运行的时间有多少秒。这个程序的关键之处是 `doit` 函数。它接受一个字符串指针和一个 `clock_t` 类型的值，然后计算并输出进程每部分实际的计时信息。

资源利用

进程的资源利用包括的内容不只是 CPU 时间。你还要考虑进程的内存使用量、内存如何进行组织、进程访问内存方式的类型、进程执行 I/O 操作的种类和数量以及进程产生的网络活动的数量和种类。内核会为每个进程跟踪所有这些信息，甚至还有更多的东西。至少内核有能力做到这一点。获得信息的结构是一个 `rusage` 结构，它在头文件 `<sys/resource.h>` 中定义。这个结构的定义如下：

```

#include <sys/resource.h>
struct rusage {
    struct timeval ru_utime;    /* user time used */
    struct timeval ru_stime;    /* system time used */
    long ru_maxrss;            /* maximum resident set size */
    long ru_maxixrss;          /* shared memory size */
    long ru_maxidrss;          /* unshared data size */
    long ru_maxisrss;          /* unshared stack size */
    long ru_minflt;            /* page reclaims */
    long ru_majflt;            /* page faults */
    long ru_nswap;             /* swaps */
    long ru_inblock;           /* block input operations */
    long ru_outblock;          /* block output operations */
    long ru_msgsnd;            /* messages sent */
    long ru_msgrcv;            /* messages received */
    long ru_nsignals;          /* signals received */
    long ru_nvcsw;             /* voluntary context switches */
    long ru_nivcsw;            /* involuntary context switches */
};

```

不幸的是，Linux 只能跟踪表 13.2 列出的资源。

表 13.2 被跟踪的系统资源

资源	描述
<code>ru_utime</code>	执行用户模式（非内核）代码的时间
<code>ru_stime</code>	执行内核代码（用户代码对系统服务的请求）的时间

(续表)

资源	描述
ru_minflt	次要失效 (minor fault) 数 (内存访问没有引起磁盘访问)
ru_majflt	主要失效 (major fault) 数 (内存访问引起磁盘访问)
ru_nswap	因主要错误而从磁盘读取的内存页数

如表 13.2 所示, 有两种类型的内存失效: 次要失效和主要失效。当 CPU 必须访问主存 (RAM) 而不是从 L2 或 L1 高速缓存 (在 x86 体系结构上分别是 level 1 或 level 2 级高速缓存)、CPU 的片上或高速缓冲存储器中读取数据时, 就发生了次要失效 (minor fault)。出现这种失效的原因是因为 CPU 需要的代码或数据不在寄存器或高速缓存中。当进程因所需代码或数据不在 RAM 中而必须从磁盘读取数据时就发生了主要失效 (major fault)。rusage 结构的成员 ru_nswap 保存了因出现主要失效而必须从磁盘读取的内存页面数量。

使用 getrusage (代表 get resource usage, 获得资源利用) 调用能够获得这些信息。成员 ru_utime 和 ru_stime 保存进程累计花费的用户和系统 CPU 时间。getrusage 提供给你而 times 没有的信息是内存失效的数目以及与失效相关的磁盘访问的数目。和函数 times 不同, 如果你想要获得父进程和子进程的信息, 必须调用两次 getrusage。getrusage 的原型如下:

```
#include <sys/times.h>
#include <sys/resource.h>
#include <unistd.h>
int getrusage(int who, struct rusage *usage);
```

usage 是指向函数要填充的 rusage 结构的指针。参数 who 决定了返回谁的资源利用信息, 是调用进程的还是它的子进程的信息。who 的取值不是 RUSAGE_SELF 就是 RUSAGE_CHILDREN。如果执行成功则 getrusage 返回 0; 如果执行出错则返回-1。

程序清单 13.5 用 getrusage 函数而不是 times 函数重写了前面的例子。

程序清单 13.5 resusg2.c

```
/*
 * resusg2.c - Getting resource usage with getrusage
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/times.h>
#include <sys/resource.h>
#include <time.h>
#include <unistd.h>

void err_quit(char *);
void doit(char *, long);

int main(void)
{
    struct rusage usage;
```

```

    /* redirect output to prevent screen clutter */
    system("grep the /usr/doc/*/* > /dev/null 2> /dev/null");

    /* get the resource structure for the parent */
    if((getrusage(RUSAGE_SELF, &usage)) == -1)
        err_quit("getrusage");

    puts("parent times");
    doit("\tuser CPU", usage.ru_utime.tv_sec);
    doit("\tsys CPU", usage.ru_stime.tv_sec);

    puts("parent memory stats");
    doit("\tminor faults", usage.ru_minflt);
    doit("\tmajor faults", usage.ru_majflt);
    doit("\tpage  swaps", usage.ru_nswap);

    /* get the resource structure for the child */
    if((getrusage(RUSAGE_CHILDREN, &usage)) == -1)
        err_quit("getrusage");

    puts("child times");
    doit("\tuser CPU", usage.ru_utime.tv_sec);
    doit("\tsys CPU", usage.ru_utime.tv_sec);

    puts("child memory stats");
    doit("\tminor faults", usage.ru_minflt);
    doit("\tmajor faults", usage.ru_majflt);
    doit("\tpage  swaps", usage.ru_nswap);

    exit(EXIT_SUCCESS);
}

void doit(char *str, long resval)
{
    printf("%s: %ld\n", str, resval);
}

void err_quit(char *str)
{
    perror(str);
    exit(EXIT_FAILURE);
}

```

这个程序执行的 `grep` 命令和前一个例子中的一样。但是，除了计时信息以外，这个程序还显示了父进程和子进程的内存使用情况。该程序示范运行的输出结果如下：

```

$ ./resusg2
parent times
    user CPU: 0
    sys CPU: 0
parent memory stats
    minor faults: 12

```

```
major faults: 92
page swaps: 0
child times
user CPU: 1
sys CPU: 1
child memory stats
minor faults: 180
major faults: 12030
page swaps: 0
```

程序示范运行的结果表明, `getrusage` 产生的计时信息和 `times` 产生的计时信息精度不一样。另一方面, 使用 `getrusage` 函数能够比较清楚地了解进程的内存使用情况。实际上, 在示范运行中出现的主要失效数目证实了前面有关该程序需要大量磁盘 I/O 操作的讨论。进程因所需数据不在内存中而从磁盘读取数据多达 12 122 次。但是没有发生页交换。

会话和进程组

当你考虑的进程彼此间没有严格的父子关系时就出现了另一类进程属性。出现这种情况后, 迄今为止所讨论的简单的父子模型就不能充分地描述进程间的相互关系。例如, 有一个打开的 `xterm` 窗口, 而你在 `xterm` 窗口中分别执行了 3 条命令: `ls`、`cat` 和 `vi`。其中运行了 3 条命令的 `xterm` 是父进程, 还是解释并执行这些命令的 `shell` 才是父进程? 从直觉上讲, 这 3 条命令显然彼此间有关系, 但是又远没有本章前面介绍的简单的父子关系那样清楚。然而, 这 3 个进程是同一会话过程的所有部分。随后将更细致地进行说明。

当在管道中执行命令 (例如, `ls -l | sort | more`) 时又出现了另一种模棱两可的情形。此时相互有关系的命令不是父子关系而是同一进程组的成员关系。图 13.1 揭示了进程、会话和进程组之间的关系。

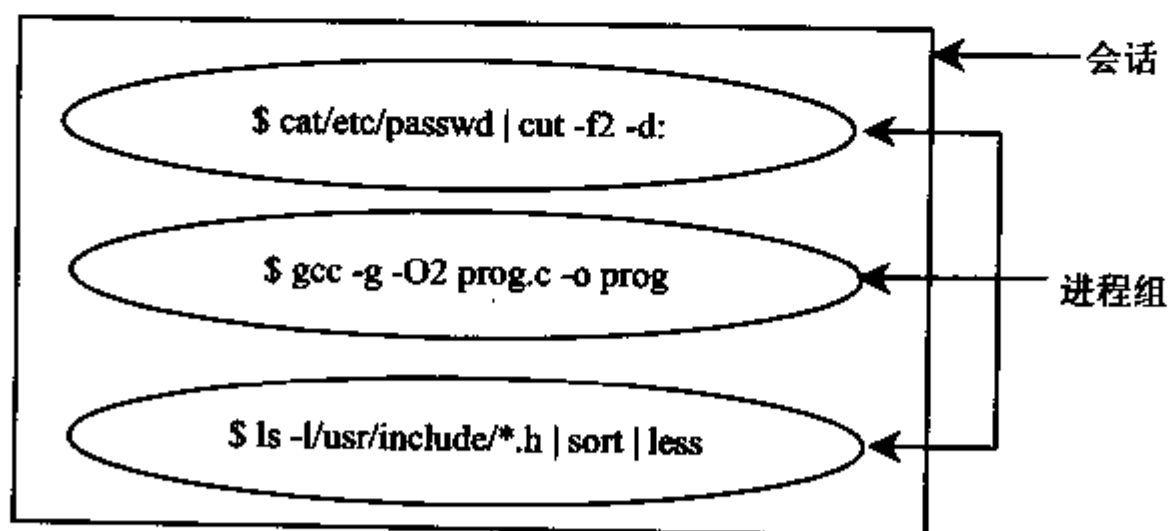


图 13.1 进程、会话和进程组

一个进程组 (process group) 是相关进程的一个集合, 这些相关进程通常是在一个管道中的命令序列。在进程组中的所有进程都具有相同的进程组号, 即 `PGID`。使用进程组的目的是为了更方便作业控制。例如, 假定你运行了命令行管道 `ls -l /usr/include | sort | wc -l`。如果要在其尚在运行时杀死它 (按 `Ctrl+C` 键), 则 `shell` 需能终止所有的进程。它通过杀死进程组而不是每一个进程来做到这一点。

会话 (session) 由一个或多个进程组构成。会话领导 (session leader) 进程是创建会话的进程。每个会话都有惟一的标识号, 称为会话 ID (session ID), 它只是会话领导进程的 PID。会话对进程组起的作用和进程组对单个进程起的作用一样。

假如你在后台执行了前面提到过的管道命令 (`ls -l /usr/include | sort | wc -l`), 并且还在前台执行其他命令。现在, 如果你正在一个 X Window 终端里运行这些命令, 并且在所有进程正在运行的时候关闭终端窗口, 则内核会向控制进程 (会话领导进程) 发送一个信号, 接着会话领导进程便逐个杀死上一段所介绍的各个进程组。

13.3 创建进程

本节介绍的调用既有系统调用也有库函数。每小节都简要介绍一个函数调用, 列出它的原型, 并且演示如何使用该函数调用。

13.3.1 使用 system 函数

system 函数的原型如下, 它通过把 system 传递给 `/bin/sh -c` 来执行 string 所指定的命令, string 中可以包含选项和参数, 接着整个命令行 (`/bin/sh -c string`) 又传递给系统调用 `execve`, 这个系统调用随后介绍。如果没有找到 `/bin/sh`, system 返回 127, 如果出现其他错误则返回 -1, 如果执行成功则返回 string 的代码。但是如果 string 为 NULL, system 返回一个非 0 值, 否则返回 0。

```
#include <stdlib.h>
int system(const char *string);
```

程序清单 13.6 给出了一个使用 system 调用的示例程序。

程序清单 13.6 system.c

```
/*
 * system.c - Demonstrate the system() call
 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int retval;

    retval = system("ls -l");

    if(retval == 127) {
        fprintf(stderr, "/bin/sh not available\n");
        exit(127);
    } else if(retval == -1) {
        perror("system");
        exit(EXIT_FAILURE);
    }
}
```

```

    } else if(retval != 0) {
        fprintf(stderr, "command returned %d\n", retval);
        perror("ls");
    } else {
        puts("command successfully executed");
    }
    exit(EXIT_SUCCESS);
}

```

执行 `make system` 编译这个程序。该程序使用 `system` 调用用 `ls -l` 创建一个目录内容的清单。示范运行的输出结果和下面类似：

```

$ ./system
total 235
-rw-r--r--      1 kwall users    1108 Jul 30 11:51 13fig01.fig
-rw-r--r--      1 kwall users   16623 Jul 30 11:51 13fig01.pcx
-rw-r--r--      1 kwall users     484 Jul 28 00:32 Makefile
-rw-r--r--      1 kwall users     180 Jul 27 20:57 abort.c
-rw-r--r--      1 kwall users     530 Jul 27 20:57 child.c
-rw-r--r--      1 kwall users     319 Jul 27 20:57 execs.c
-rw-r--r--      1 kwall users     319 Jul 27 20:57 execve.c
-rw-r--r--      1 kwall users     779 Jul 27 23:55 getname.c
-rw-r--r--      1 kwall users     331 Jul 27 20:57 ids.c
-rw-r--r--      1 kwall users     899 Jul 27 20:57 killer.c
-rw-r--r--      1 kwall users     217 Jul 27 20:57 prpids.c
-rw-r--r--      1 kwall users     889 Jul 28 00:40 resusg1.c
-rw-r--r--      1 kwall users    1423 Jul 28 01:32 resusg2.c
-rwxr-xr-x      1 kwall users    5491 Jul 30 12:04 system
-rw-r--r--      1 kwall users     470 Jul 27 21:14 system.c
-rw-r--r--      1 kwall users     532 Jul 27 20:57 testenv.c
-rw-r--r--      1 kwall users     686 Jul 27 20:57 waiter.c
command successfully executed

```

13.3.2 fork 系统调用

`fork` 调用创建一个新进程。新的进程或者说子进程是调用进程或者说父进程的副本。`fork` 的语法是

```

#include <unistd.h>
pid_t fork(void);

```

如果 `fork` 执行成功，就向父进程返回子进程的 `PID`，并向子进程返回 `0`。这意味着即使你只调用 `fork` 一次，它也会返回两次。

`fork` 创建的新进程是和父进程（除了 `PID` 和 `PPID`）一样的副本，包括真实和有效 `UID` 和 `GID`、进程组和会话 `ID`、环境、资源限制、打开的文件以及共享内存段。

父进程和子进程之间有一点区别。子进程没有继承父进程的超时设置（使用 `alarm` 调

用)、父进程创建的文件锁,或者未决信号。要理解的关键概念是 `fork` 创建的新进程是父进程的一个准确副本。

程序清单 13.7 展示了一个使用 `fork` 的简单示例程序。

程序清单 13.7 `child.c`

```
/*
 * child.c - Simple fork usage
 */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    pid_t child;

    if((child = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if(child == 0) {
        puts("in child");
        printf("\tchild pid = %d\n", getpid());
        printf("\tchild ppid = %d\n", getppid());
        exit(EXIT_SUCCESS);
    } else {
        puts("in parent");
        printf("\tparent pid = %d\n", getpid());
        printf("\tparent ppid = %d\n", getppid());
    }
    exit(EXIT_SUCCESS);
}
```

执行命令 `make child` 编译该程序。这个程序的输出应该和下面类似:

```
$/child
in parent
in child
    child pid = 14091
    child ppid = 14090
    parent pid = 14090
    parent ppid = 1549
```

正如你从输出结果中所看到的那样,子进程的 PPID(父进程 ID)和父进程的 PID 一样,都是 14090。输出结果还揭示了有关 `fork` 用法的关键一点:你不能预计父进程是在它的子进程之前还是之后运行。这可以通过输出结果的奇怪形式看出来。输出的第一行来自父进程,第二到第四行来自子进程,而第五和第六行又来自父进程。它的执行是无序的,也就是说,是异步的。

fork 的异步行为意味着你不应该在子进程中执行依赖于父进程的代码，反之亦然。这样做会导致出现潜在的竞态条件（race condition），当多个进程要使用共享资源但访问又依赖于进程执行的顺序时就会发生竞态条件。竞态条件很难捕获，因为绝大多数时间都是产生竞态条件的代码在执行。竞态条件潜在的影响难以预测，但是症状可能包括无法预计的程序行为、明显的系统挂起、在轻负荷的系统上系统响应缓慢，或者干脆发生系统崩溃。

fork 调用可能失败，原因可以是系统上已经运行了太多的进程，也可以是试图 fork 的 UID 已经超过了允许它执行的进程数。如果 fork 执行失败，它会向父进程返回-1，并且不创建子进程。示例程序通过检查 fork 返回值避免这种情况。检查还让程序判断出它是在子进程还是在父进程中。

注意： fork 过程包括把父进程的全部内存映像复制给子进程。这是一种天生缓慢的过程，所以 UNIX 的设计者创建了 vfork 调用。vfork 也创建新进程，但它不产生父进程的副本。而在调用 exec 或 exit 之前，新进程运行在父进程的地址空间中——如果它访问任何父进程的内存，那部分内存才复制给子进程。这个特点称为写时复制（copy-on-write）。

vfork 背后的基本原理是加速创建新进程。另外，vfork 还有额外的特性来确保子进程在父进程之前先运行，因而减少了竞态条件的威胁。但是在 Linux 下，vfork 只是 fork 的包裹函数，因为 Linux 已经使用了写时复制的技术。因此 Linux 的 fork 和 UNIX 的 vfork 一样快，但是 Linux 的 vfork 因为是 fork 的别名，它不能保证子进程在父进程之前运行。

13.3.3 exec 函数族

exec 函数实际上是包含了 6 个函数的函数族，这 6 个函数中的每一个都在调用规则和用法上略有区别。无论其功能的多样化如何，还是习惯上称它们为 exec 函数。和 fork 类似，exec 也在<unistd.h>中声明。它的原型为：

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execlx(const char *path, const char *arg, char *const envp[]);
int execv(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execvp(const char *file, char *const argv[]);
```

exec 用被执行的程序完全替换了调用进程的映像。fork 创建一个新进程就产生了一个新的 PID，exec 启动一个新程序，替换原有的进程。因此，被执行进程的 PID 不会改变。

execve 接受 3 个参数：path、argv 和 envp。path 是要执行的二进制文件或脚本的完整路径。argv 是要传递给程序的完整参数列表，包括 argv[0]，它一般是执行程序的名字。envp 是指向用于执行 execed 程序的专门环境的指针（在示例程序中为 NULL）。

程序清单 13.8 使用 execve 在当前目录下执行一条 ls 命令。

程序清单 13.8 execve.c

```
/*
 * execve.c - Illustrate execve
 */
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *args[] = {"/bin/ls", NULL};

    if(execve("/bin/ls", args, NULL) == -1) {
        perror("execve");
        exit(EXIT_FAILURE);
    }

    puts("shouldn't get here");

    exit(EXIT_SUCCESS);
}
```

试验运行一下这个程序（用 `make execve` 编译），产生的输出如下：

```
$ ./execve
13fig01.fig Makefile child.c getname.c oldcode resusg1.c system.c
13fig01.pcx abort.c execve ids.c out resusg2.c testenv.c
607211x.doc child execve.c killer.c prpids.c system waiter.c
```

正如你从输出中所看到的那样，`puts` 语句没有执行。为什么呢？如果 `exec` 执行成功，它不会返回调用进程。这样才有意义，因为前面提到过，`exec` 完全用新程序替换了调用进程，所以不会留下调用进程的任何痕迹。也就是说，不再有能返回的调用进程了。但是如果 `exec` 执行失败，它就返回-1 并且设置全局变量 `errno`。

因为 `exec` 函数族中的 6 个函数具有易混淆的相似性，所以下面对它们的语法、行为、相似性和区别给予完整的讨论。

四个函数——`execl`、`execv`、`execle` 和 `execve`——第一个参数都是路径名。`execlp` 和 `execvp` 的第一个参数则是文件名，如果文件名没有包含“/”，它们会模仿 `shell` 的行为搜索 `$PATH` 找到要执行的二进制文件。

三个名字中含有 `l` 的函数希望接收以逗号分隔的参数列表，列表以 `NULL` 指针作为结束标志，这些参数将传递给被执行的程序。但是，名字中包含 `v` 的函数则接收一个向量，也就是指向以空结尾的字符串的指针数组。这个数组必须以一个 `NULL` 指针作为结束标志。例如，假设你希望执行命令 `/bin/cat /etc/passwd /etc/group`。如果使用带 `l` 的函数之一，则只需把这些值的每一个作为参数，再以 `NULL` 作为结束标志传递给该函数即可，如下所示：

```
execl("/bin/cat", "/bin/cat", "/etc/passwd", "/etc/group", NULL);
```

但如果用带 `v` 的函数之一，必须先构造一个 `argv` 数组，然后把这个数组传递给 `exec` 函数。你的代码应该和下面的类似：

```
char *argv[] = {"/bin/cat", "/etc/passwd", "/etc/group", NULL};
execv("/bin/cat", argv);
```

最后，两个以 `e` 结尾的函数——`execve` 和 `execle`——可以让你为被执行的程序创建专门的环境。这个环境保存在 `envp` 中，它也是一个指向以空结尾的字符串数组的指针，数组中每个字符串也是以空结尾。每个字符串的形式为“`name=value`”对，`name` 是环境变量的名字而 `value` 是它的值。例如

```
char *envp[] = {"PATH=/bin:/usr/bin", "USR=joeblow", NULL};
```

在这个例子里，`PATH` 和 `USER` 是环境变量名，而 `/bin:/usr/bin` 和 `joeblow` 是变量的值。

其他 4 个函数隐式地通过全局变量 `environ` 接受它们的环境，`environ` 是一个指向字符串数组的指针，数组中包含了调用进程的环境。使用 `putenv` 和 `getenv` 函数可以操控这些函数继承的环境。它们的原型如下：

```
#include <stdlib.h>
int putenv(const char *string);
char *getenv(const char *name);
```

`getenv` 查找名为 `name` 的环境变量并返回指向其值的指针，如果没有找到则返回 `NULL`。`putenv` 添加或改变 `string` 中指定的“`name = value`”对。如果它执行成功，则返回 0。如果它执行失败，则返回 -1。使用 `getenv` 和 `putenv` 编写的代码和下面的类似：

```
char envval[] = {"MYPATH=/user/local/someapp/bin"};
if(putenv(envval) == 0)
    puts("putenv succeeded");

if (getenv("MYPATH"))
    printf("MYPATH=%s\n",getenv("MYPATH"));
else
    puts("MYPATH unassigned");
```

13.3.4 使用 `popen` 函数

`popen` 函数的行为和 `system` 函数类似，它是一种无需使用 `fork` 和 `exec` 就能在执行外部程序的简易方法。但是和 `system` 不同，`popen` 使用管道来工作。它的原型为

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

`popen` 调用管道，并创建通向标准输入，或者从 `command` 指定的程序或脚本的标准输出来的管道，但不是同时两者都有。第二个参数 `type` 在读取管道的 `stdout` 时为 `r`，在写入 `stdin` 时为 `w`。注意，`popen` 的 I/O 用法有点儿违反直觉。读和写都是相对于 `command` 面言的，所以 `command` 的输出是从 `stdout` 读入的。要向 `command` 输入，则需向它的 `stdin` 写入。

13.4 控制进程

在某些场合下,用 `fork` 或 `exec` 简单地创建一个新进程就是你要完成的全部工作。但是,在另外一些场合下,可能你还要设置某些进程属性,或者改变产生的进程的行为。本节就介绍实现上述功能的函数调用。

13.4.1 等待进程——wait 函数族

一旦你用 `fork` 或 `exec` 创建了一个新进程,为了收集新进程的退出状态并防止出现僵进程 (zombie process),父进程应该等待新进程终止。对于 `exec` 来说,你可以使用多种函数。但是为了避免出现混乱,本节只集中介绍 `wait` 和 `waitpid` 函数。

什么是僵进程?一个僵进程是在父进程有机会用 `wait` 或 `waitpid` 收集它的退出状态之前就终止的子进程。父进程通过使用 `wait` 函数之一检索内核的进程表取得退出状态来收集 (collect) 子进程的退出状态。一个进程之所以称为僵进程是因为它虽然死掉了,但依然在进程表中存在。子进程退出后,分配给它的内存和其他资源都被释放,但是它还在内核的进程表中保留了一条。内核在父进程收回子进程的退出状态之前一直保留着它。

有一两个僵进程不算什么问题,但是如果一个程序频繁执行 `fork` 和 `exec` 却又不能收集退出状态,那么最终会填满进程表,这会影响性能而且必要时会导致系统重新自举——这种情况不希望在关键应用环境中出现。

另一方面,一个孤儿进程 (orphan process) 是一个父进程在调用 `wait` 或 `waitpid` 之前就已经退出的子进程。此时, `init` 进程成为子进程的父进程并且收集它的退出状态,从而避免出现僵进程。

使用 `wait` 或 `waitpid` 调用可以收集子进程的退出状态。它们的原型如下:

```
#include <sys/wait.h>
#include <sys/types.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

`status` 保存子进程的退出状态。`pid` 是等待进程的 PID。它能接受表 13.3 列出的 4 种取值中的一个。

表 13.3 pid 的可能值

值	描述
-1	等待任何 PGID 等于 PID 的绝对值的子进程
1	等待任何子进程
0	等待任何 PGID 等于调用进程的子进程
>0	等待 PID 等于 pid 的子进程

`options` 规定 `wait` 调用的行为应该如何。它可以是 `WNOHANG`, 导致 `waitpid` 在没有子进程退出时立即返回,也可以是 `WUNTRACED`, 意味着它应该因为存在没有报告状态的进程而返回。你也可以对它们执行逻辑“或”(OR)操作,取得两种行为(也就是说,给 `options` 参数传送 `WNOHANG || WUNTRACED`)。程序清单 13.9 演示了 `waitpid` 用法。

程序清单 13.9 使用 waitpid

```
/*
 * waiter.c - Simple wait usage
 */
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    pid_t child;
    int status;

    if((child = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if(child == 0) {
        puts("in child");
        printf("\tchild pid = %d\n", getpid());
        printf("\tchild ppid = %d\n", getppid());
        exit(EXIT_SUCCESS);
    } else {
        waitpid(child, &status, 0);
        puts("in parent");
        printf("\tparent pid = %d\n", getpid());
        printf("\tparent ppid = %d\n", getppid());
        printf("\tchild exited with %d\n", status);
    }
    exit(EXIT_SUCCESS);
}
```

使用 `make waiter` 编译这个程序。`waiter` 的输出和下面类似:

```
$ ./waiter
in child
    child pid = 14182
    child ppid = 14181
in parent
    parent pid = 14181
    parent ppid = 1549
    child exited with 0
```

这个程序和 `child.c` 非常相似。它专门等待 `child` 指定的子进程返回, 并且显示子进程的退出状态。父进程和子进程的输出没有像 `child.c` 那样混合起来, 因为父进程直到子进程退出才停止执行。`Waitpid` (和 `wait`) 返回退出的子进程 PID, 如果在 `options` 中指定 `WNOHANG` 则返回 0, 如果出错则返回 -1。

13.4.2 杀死程序

一个进程由于以下 5 个原因中的一个而终止：

- 它的 `main` 函数调用了 `return`
- 它调用了 `exit`
- 它调用了 `_exit`
- 它调用了 `abort`
- 它被一个信号终止

前 3 个理由是正常终止，而后两个则是非正常终止。但是无论进程为何终止，最后都执行相同的内核代码、关闭打开的文件、释放内存资源，并且执行其他要求的清理工作。因为本书假定读者有 C 编程能力，所以应该不需要解释 `return` 函数。

exit 函数

读者已经看到，`exit` 函数作为 C 标准库的一部分在本章的示例程序中贯穿使用。这里不再举例说明该函数的用法，这个函数在 `<stdlib.h>` 中声明的原型如下：

```
int exit(int status);
```

`exit` 导致程序正常终止并且返回父进程的状态 (`status`)。用 `atexit` 登记的函数也被执行。`_exit` 函数在 `<unistd.h>` 中声明。它立即终止调用它的进程：用 `atexit` 登记的函数不被执行。

使用 abort 函数

如果你需要异常地终止一个程序，可以使用 `abort` 函数。在 Linux 下，`abort` 还可让程序产生内存转储 (`core dump`)，这是大多数调试器用于分析程序崩溃时的文件。虽然任何打开的文件都被关闭了，但 `abort` 函数仍然是个粗暴的调用，应该作为最后的手段来使用，比如当你碰到类似严重内存不足这样的错误，无法用程序的方法处理时再用。`abort` 也是一个标准库函数。它原型为

```
#include <stdlib.h>
void abort(void);
```

程序清单 13.10 中的程序显示了 `abort` 函数怎样运行。

程序清单 13.10 abort.c

```
/*
 * abort.c - Demonstrate the abort system call
 */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    abort();
}
```

```

    /* Shouldn't get here */
    exit(EXIT_SUCCESS);
}

```

使用 `make abort` 编译这个程序。它的输出取决于你的系统如何配置以处理内存转储文件，应该和下面类似：

```

$ ./abort
Aborted
$ ulimit -c unlimited
$ ./abort
Aborted (core dumped)

```

注意，你的系统可能不能生成一个 `core` 文件。如果它没有生成 `core` 文件，则按照上面的示范运行来使用 `shell` 的命令 `ulimit`。

使用 `kill` 函数

前面两小节集中介绍进程如何杀死自己。一个进程能够用 `kill` 函数杀死另一个进程，它的原型如下：

```

#include <signal.h>
#include <sys/types.h>
int kill(pid_t pid, int sig);

```

`pid` 指定了要杀死的进程，而 `sig` 是要发送的信号。因为本节的内容是杀死一个进程，所以只需要关注 `SIGKILL` 这一种信号。13.5 节“信号”将扩展对信号的讨论。现在就先了解这么多吧。

程序清单 13.11 展示了如何杀死一个进程。执行 `make killer` 编译这个程序。

程序清单 13.11 `killer.c`

```

/*
 * killer.c - Killing other processes
 */
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    pid_t child;
    int status, retval;

    if((child = fork()) < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
}

```



```
    if(child == 0) {
        /* Sleep long enough to be killed */
        sleep(1000);
        exit(EXIT_SUCCESS);
    } else {
        /* Use WNOHANG so wait will return */
        if((waitpid(child, &status, WNOHANG)) == 0) {
            retval = kill(child, SIGKILL);
            if(retval) {
                /* Kill failed, so wait on child to exit */
                puts("kill failed\n");
                perror("kill");
                waitpid(child, &status, 0);
            } else
                printf("%d killed\n", child);
        }
    }
    exit(EXIT_SUCCESS);
}
```

这个程序的输出类似下面：

```
$ ./killer
14215 killed
```

在确信 `fork` 执行成功后，子进程睡眠 1000 秒钟，然后退出。同时父进程在子进程上调用 `waitpid`，但它使用 `WNOHANG` 选项，所以调用会立即返回。父进程接着杀死子进程。如果 `kill` 执行失败，它返回 -1；否则它返回 0。如果 `kill` 失败，父进程第二次调用 `waitpid`，保证它在子进程退出后再停止执行。否则，父进程显示一条成功消息然后退出。`kill` 常常用来终止一个进程或进程组，但是它也能用来向进程或进程组发送任何信号。

13.5 信 号

信号是由相同或不同的进程向一个进程传递的事件。信号通常用来向一个进程通知异常事件。

13.5.1 什么是信号

信号是硬件中断的软件模拟，进程正在执行时，几乎在任何时刻都会发生事件。这种不可预测性意味着信号是异步的。不但信号可在任何时刻发生，当信号发出时接收信号的进程也可以没有控制权。每个信号名都以 `SIG` 开头，比如 `SIGTERM` 或 `SIGHUP`。这些名字对应于正整数常量，称为信号量 (signal number)，它们在系统的头文件 `<signal.h>` 中定义。

许多情况下都会出现信号。硬件异常，如非法的内存引用，就能产生一个信号。软件异常，如试图向一个没有读取的管道执行写操作 (`SIGPIPE`)，也会产生一个信号。前面讨

论的 `kill` 函数向被杀死的进程发出一个信号，就和 `kill` 命令一样。最后，由终端产生的操作，如键入 `Ctrl+Z` 以挂起前台进程，也产生信号。

当进程收到一个信号后，它可以对信号采取如下三种措施之一：

- 忽略这个信号。
- 捕获 (`trap/catch`) 这个信号，这将导致执行一段称为信号处理器的特殊代码。这叫作处理信号。
- 允许执行信号的默认操作。

在继续下面的内容之前，你需要理解一些在讨论信号时常用的术语。当导致信号发生的事件出现时，比如硬件异常，就产生 (`generate`) 一个针对某个进程的信号。相反地，当进程对发送给它的信号采取措施时，就称该信号被递送 (`deliver`)。在产生信号和递送信号之间的时间间隔称为信号未决 (`pending`)。递送信号可以被阻塞或被延迟。这个信号一直被延迟，直到被解除阻塞或者接到进程对该信号的部署方式改变为忽略 (`ignore`) 为止。信号的部署 (`disposition`) 是指进程如何响应信号。一个进程可以忽略信号、允许其默认的操作发生，或者处理该信号，这意味着执行对应此信号的自定义代码。一个被阻塞的信号也称为未决的信号。一个信号集合 (`signal set`) 是一个 C 数据类型 `sigset_t`，它在 `<signal.h>` 中定义，能够表示多种信号。最后，一个进程的信号掩码 (`mask`) 是进程目前正阻塞不能递送的信号集合。

13.5.2 发送信号

从编程的角度看，向正在运行的程序发送信号的方法有两种：使用 `kill` 命令 (`kill(1)`) 和使用 `kill(2)` 函数。`kill` 命令实际上是 `kill` 函数的用户接口。

使用 `kill` 命令

要在程序中使用 `kill` 命令，必须调用 `system`、`fork` 或 `exec`。正如你在本章前面所学到的那样，前面的两个调用产生一个新进程来执行 `kill`，而 `exec` 在运行 `kill` 之前替换了调用进程。但是，它们的结果是一样的：目标进程被终止。

使用 `kill` 函数

使用 `kill` 函数比用 `exec` 调用执行 `kill` 命令要简单，因为不必有准备 `exec` 字符串的额外步骤。你所需要的只是 `PID` 和要用的信号。

你可能会注意到没有哪个信号的值为 0。值为 0 的信号是一个空信号 (`null signal`)，它有特殊的用途。如果你传递给 `kill` 函数空信号，它根本不会发出这个信号，但它会执行正常的出错检查。如果你要通过查看进程的 `PID` 来判断一个进程是否正在运行，那么这就会派上用场。但是请记住，`PID` 会周期性地循环使用，那么在一个繁忙的机器上这种测试某个进程是否存在的方法并不可靠。

程序清单 13.12 使用 `kill` 函数向一个睡眠中的子进程发送两个信号：一个信号被忽略，而另一个信号杀死进程。使用 `make fkill` 命令编译该程序。

程序清单 13.12 fkill.c

```
/*
 * fkill.c - Send a signal using kill(2)
 */
#include <sys/types.h>
#include <wait.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    pid_t child;
    int errret;

    if((child = fork()) < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if(child == 0) { /* in the child process */
        sleep(30);
    } else { /* in the parent */
        /* send a signal that gets ignored */
        printf("sending SIGCHLD to %d\n", child);
        errret = kill(child, SIGCHLD);
        if(errret < 0)
            perror("kill:SIGCHLD");
        else
            printf("%d still alive\n", child);
        /* now kill the child process */
        printf("killing %d\n", child);
        if((kill(child, SIGTERM)) < 0)
            perror("kill:SIGTERM");
        /* have to wait to reap the status */
        waitpid(child, NULL, 0);
    }
    exit(EXIT_SUCCESS);
}
```

下面是该程序两次运行的结果:

```
$ ./fkill
sending SIGCHLD to 14241
14241 is still alive
killing 14241
$ ./fkill
sending SIGCHLD to 14243
14243 is still alive
killing 14243
```

要注意的第一点是，和看到的情况一样让人觉得奇怪，kill 能够用来发送除了杀死一个进程（SIGKILL、SIGTERM、SIGQUIT）之外的其他信号。第二，实际上子进程将忽略 SIGCHLD 信号。因为我的系统相当不活跃，所以我可以使用空信号来确认发给信号的进程还活着。最后，SIGTERM 信号能够终止子进程。

调用 waitpid 是能够避免 kill 失败的安全方法。正如前面讨论 fork 调用时所说的那样，预先没有办法知道子进程是在父进程之前还是之后终止。如果父进程先退出，子进程就变成孤儿而被 init 接管，随后 init 会接收子进程的退出状态。但如果子进程先死掉，父进程必须接收它的退出状态以避免它变成浪费内核进程表项的僵进程。

13.5.3 捕获信号

捕获并处理信号是和发送信号相对的另一方面。每个进程都能决定怎样响应除了 SIGSTOP 和 SIGKILL 之外的其他所有信号，而这两个信号不能被捕获或者忽略。捕获信号最简单的方法不是真去捕获信号而是等待它们被发送过来。alarm 函数设置了一个定时器，当定时器时间到时就发送 SIGALRM 信号。pause 函数也有类似功能，但它是把进程挂起直至进程收到任何信号。

设置超时

原型在<unistd.h>中 alarm 函数在调用进程中设置一个定时器。当定时器时间到时，它就向调用进程发出 SIGALRM 信号，除非调用进程捕获这个信号，否则 SIGALRM 的默认动作是中止进程。alarm 的原型为

```
#include<unistd.h>
unsigned int alarm(unsigned int seconds);
```

seconds 是计时器时间到后时钟的秒数。如果没有设置其他超时，则返回值为 0，否则返回值为前面安排的超时中保留的秒数。一个进程只能设置一次超时。把 seconds 设为 0 就会取消前面的超时设置。

使用 pause 函数

pause 函数挂起调用它的进程直至有任何信号达到。调用进程必须有能力处理递送到的信号，否则信号的默认部署就会发生。pause 的原型如下：

```
#include <unistd.h>
int pause(void);
```

只有进程捕获到一个信号时，pause 才返回调用进程。如果递送到的信号引发了对信号的处理，那么处理工作将在 pause 返回前执行。pause 总是返回-1 并且把变量 errno 设置为 EINTR。

定义一个信号处理器

在某些情况下，一个信号的默认动作就是所希望的行为。但是在更多的场合下，你可能想要改变默认行为或者执行额外的任务。在这种情况下，必须定义并安装一个自定义的信号处理器以覆盖默认的动作。

考虑这样的情况，一个父进程产生了好几个子进程。当子进程终止时，父进程接到 SIGCHLD 信号。为了跟踪它的子进程并减少它们的退出状态，父进程在产生每个子进程后立即调用 `wait`，或者更高效的是建立一个信号处理器，在它每次接收到 SIGCHLD 信号时调用 `wait`（或者 `waitpid`）。

POSIX 定义一组创建并处理信号的函数。一般的步骤是先创建一个信号集合、设置想要捕获的信号，再向内核登记一个信号处理器，然后等候捕捉信号。

可以使用下面 5 个函数创建、设置以及查询一个信号集合，它们都在 `<signal.h>` 中定义：

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

`set` 是一个 `sigset_t` 类型的信号集合，这在本节的开头已经解释过了。`sigemptyset` 初始化信号集合 `set`，从集合中去除所有的信号。相反地，`sigfillset` 初始化信号集合，在集合中包含所有的信号。`sigaddset` 把信号 `signum` 添加到信号集合中。`sigdelset` 从集合中删除信号 `signum`。

这 4 个函数执行成功后都返回 0，出错时则返回 -1。最后，`sigismember` 检测 `signum` 是否在信号集合中，如果在就返回 1（真），不在则返回 0（假）。

创建一个信号集合

使用 `sigemptyset` 或 `sigfillset` 初始化一个信号集合。如果你创建一个空的信号集合，需要使用 `sigaddset` 向感兴趣的集合中加入信号。如果你创建一个满的信号集合，需要使用 `sigdelset` 从信号掩码中删除信号。

程序清单 13.13 mkset.c

```
/*
 * mkset.c - Create a signal set
 */
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

void err_quit(char *);

int main(void)
{
    sigset_t newset;

    /* Create the set */
    if((sigemptyset(&newset)) < 0)
        err_quit("sigemptyset");
    /* Add SIGCHLD to the set */
    if((sigaddset(&newset, SIGCHLD)) < 0)
        err_quit("sigaddset");
```

```

    /* Check the signal mask */
    if(sigismember(&newset, SIGCHLD))
        puts("SIGCHLD is in signal mask");
    else
        puts("SIGCHLD not in signal mask");
    /* SIGTERM shouldn't be there */
    if(sigismember(&newset, SIGTERM))
        puts("SIGTERM in signal mask");
    else
        puts("SIGTERM not in signal mask");

    exit(EXIT_SUCCESS);
}

void err_quit(char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}

```

执行命令 `make mkset` 编译这个程序。示范运行的结果与下面类似：

```

$ ./mkset
SIGCHLD is in signal mask
SIGTERM not in signal mask

```

`mkset` 首先创建一个空的信号集合，然后把 `sigset_t` 结构的地址传递给 `sigemptyset`。接着它把 `SIGCHLD` 加入到进程的信号掩码中。最后，程序使用 `sigismember` 确认 `SIGCHLD` 是集合的一部分，而 `SIGTERM` 不是掩码的一部分。

登记一个信号处理器

简单地创建一个信号集合并添加或删除信号并没有创建一个信号处理器，也不能让你捕获或阻塞信号。还有更多的步骤要实施。首先，必须使用 `sigprocmask` 设置或修改当前的信号掩码——如果还没有设置一个信号掩码，那么所有的信号都具有它们默认的部署。一旦你设置了信号掩码，就需要使用 `sigaction` 调用为信号或要捕获的信号登记一个信号处理器。

`sigprocmask` 的原型为：

```

#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

```

根据 `how` 的取值不同，`sigprocmask` 设置或检查当前的信号掩码，`how` 可以取下面的几个值：

- `SIG_BLOCK`——`set` 包含其他要阻塞的信号
- `SIG_UNBLOCK`——`set` 包含要解除阻塞的信号
- `SIG_SETMASK`——`set` 包含新的信号掩码

如果 `how` 为 `NULL`，它就被忽略。如果 `set` 为 `NULL`，当前的掩码就保存在 `oldset` 中；如果 `oldset` 为 `NULL`，它也被忽略。`sigprocmask` 执行成功返回 0，执行失败返回 -1。

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct
               sigaction *oldact);
```

`sigaction` 为 `signum` 所指定的信号设置信号处理器。`sigaction` 结构 (`act` 和 `oldact`) 描述了信号的部署。它在 `<signal.h>` (你也猜得出来) 中的完整定义为：

```
struct sigaction{
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

`sa_handler` 是函数指针，这个函数规定了当 `signum` 中的信号产生后，要调用的处理器或者函数。这个函数必须定义为返回 `void` 类型并接受一个 `int` 类型的参数。另一种选择是，`sa_handler` 也可以是 `SIG_DFL` (它引起 `signum` 的默认动作发生)，或者是 `SIG_IGN` (它忽略 `signum` 这个信号)。

当执行信号的处理器时，触发它的信号则被阻塞。`sa_mask` 定义了在执行处理器期间应该阻塞的其他信号集合的信号掩码。`sa_flags` 是修正 `sa_handler` 行为的掩码。它可以是下面几个值之一：

- **SA_NOCLDSTOP**——进程忽略子进程产生的任何 `SIGSTOP`、`SIGTSTP`、`SIGTTIN` 和 `SIGTTOU` 信号。
- **SA_ONESHOT** 或 **SA_RESETHAND**——登记的自定义信号处理器只执行一次。在执行完毕后，恢复信号的默认动作。
- **SA_RESTART**——让可重启的系统调用起作用。
- **SA_NOMASK** 或 **SA_NODEFER**——不避免在信号自己的处理器中接收信号本身。

忽略 `sa_restorer` 元素：它已被废弃不再使用。

现在做一下总结，`sigprocmask` 控制你要阻塞的信号集合并且查询当前的信号掩码。函数 `sigaction` 向内核为一个或多个信号登记信号处理器并且设置处理器的准确行为。程序清单 13.14 只是阻塞 `SIGALRM` 和 `SIGTERM` 信号，但没有安装执行特殊操作的处理器。

程序清单 13.14 block.c

```
/*
 * block.c - Blocking signals
 */
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
```

```
void err_quit(char *);

int main(void)
{
    sigset_t newset;

    /* Create the set */
    if((sigemptyset(&newset)) < 0) err_quit("sigemptyset");
    /* Adding SIGTERM and SIGALRM */
    if((sigaddset(&newset, SIGTERM)) < 0) err_quit
        ("sigaddset:SIGTERM");
    if((sigaddset(&newset, SIGALRM)) < 0) err_quit
        ("sigaddset:SIGNALRM");

    /* Block the signals without handling them */
    if((sigprocmask(SIG_BLOCK, &newset, NULL)) < 0)
        err_quit("sigprocmask");

    /* Wait for a signal */
    pause();

    exit(EXIT_SUCCESS);
}

void err_quit(char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}
```

使用 `make block` 编译这个程序。要测试这个程序，可以在一个窗口中运行 `block` 并从另一个窗口发送信号。从第二个窗口发出的命令显示的时候有括号括起来。

```
$ ./block
[$ kill -TERM $(pidof ./block)]
[$ kill -ALRM $(pidof ./block)]
[$ kill -QUIT $(pidof ./block)]
Quit (core dumped)
```

正如你从输出结果中所看到的那样，向进程发送 `SIGTERM` 和 `SIGALRM` 不起作用，即使这两个信号的默认动作一般都是中止进程。命令 `pidof ./block` 返回和 `./block` 相关的 PID，而 `$(...)` 结构替换命令的执行结果并把它传递给 `kill` 命令。当进程接收到 `SIGQUIT` 时，如输出结果所显示的那样，进程就退出了。注意，因为程序阻塞了 `SIGTERM` 和 `SIGALRM`，所以 `pause` 函数永远不会返回，因为进程不会接受信号。

提示： `SIGUSR1` 和 `SIGUSR2` 特别保留下来作为程序员自定义的信号。你应该用它们去实现特殊的行为，比如重读一个配置文件，而不要用一个信号处理器重新定义原有的标准信号，因为它们的语义已经定义好了。

13.5.4 检测信号

`sigpending` 允许进程检测未决信号（当阻塞时被挂起的信号），然后决定是忽略它们还是递送它们。为什么要找出什么信号是未决的呢？假如你想要向一个文件写入数据，为了保持文件的完整性，写操作必须不能中断。在写入期间，你想要阻塞 `SIGTERM` 和 `SIGQUIT`，但一般情况下你不是处理它们就是许可它们的默认动作。在开始执行写入操作之前，你阻塞了 `SIGTERM` 和 `SIGQUIT` 信号。一旦写入操作成功完成，你就要检查未决信号，而如果有未决的 `SIGTERM` 或 `SIGQUIT` 信号，就要解除对它们的阻塞。或者，你也可以简单地解除对它们的阻塞而不必费力地检查它们是否未决。是否要检查未决信号完全由你来决定。如果你想要在某个信号挂起时执行一段特殊的代码，那么就要检查未决信号。否则，只需解除阻塞即可。

和其他信号函数一样，`sigpending` 的原型也在 `<signal.h>` 中定义。`sigpending` 的原型如下：

```
#include <signal.h>
int sigpending(sigset_t *set);
```

未决信号的集合在 `set` 中返回。调用本身在成功时返回 0，出错时返回 -1。使用 `sigismember` 判断你感兴趣的信号是否是未决信号，也就是说，判断它们是否在 `set` 中。

为了达到演示目的，程序清单 13.15 阻塞了 `SIGTERM` 信号，判断它是否是未决信号，然后忽略它并正常终止。

程序清单 13.15 `pending.c`

```
/*
 * pending.c - Fun with sigpending
 */
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    sigset_t set, pendset;
    struct sigaction action;

    sigemptyset(&set);

    /* Add the interesting signal */
    sigaddset(&set, SIGTERM);
    /* Block the signal */
    sigprocmask(SIG_BLOCK, &set, NULL);

    /* Send SIGTERM to myself */
    kill(getpid(), SIGTERM);

    /* Get pending signals */
    sigpending(&pendset);
```

```

/* If SIGTERM pending, ignore it */
if(sigismember(&pendset, SIGTERM)) {
    sigemptyset(&action.sa_mask);
    action.sa_handler = SIG_IGN; /* Ignore SIGTERM */
    sigaction(SIGTERM, &action, NULL);
}

/* Unblock SIGTERM */

sigprocmask(SIG_UNBLOCK, &set, NULL);

exit(EXIT_SUCCESS);
}

```

为了使程序简洁，所以省略了对出错的检查。代码的含义很直观。它创建了阻塞 SIGTERM 的信号掩码，然后使用 kill 向自己发出 SIGTERM 信号。因为信号被阻塞，所以信号不被递送。你可以使用 sigpending 和 sigismember 来判断 SIGTERM 是否未决，如果是这样，则把它的部署设置为 SIG_IGN。当你解除阻塞后，SIGTERM 不被递送，程序正常终止。

13.6 进程调度

本节讨论的调用所操作的参数能够控制和进程相关联的调度算法和优先级。关键的函数调用都在下面列出，所有的函数原型都在<sched.h>中声明：

```

#include <sched.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>
int sched_setscheduler(pid_t pid, int policy, const struct
                        sched_param *p);
int sched_getscheduler(pid_t pid);
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
int nice(int inc);

```

具有更高静态优先级的进程总是会抢先于较低静态优先级的程序而执行。对于传统的调度算法来说，具有静态优先级 0 的进程是按照它们的动态优先级来分配 CPU 时间的，动态优先级也称为它们的“谦让度 (nice)”值。

系统调用 sched_setscheduler 和 sched_getscheduler 分别用于设置或取得与某个特定进程相关的调度策略和参数（只能设置）。这些函数接收进程的 PID，即 pid，标识出要执行操作的进程：调用进程必须拥有对指定进程进行操作的权限。调度策略，即 policy 可以是 SCHED_OTHER（默认策略）、SCHED_FIFO 或 SCHED_RR 中的一个；后两个值是专门用

于对时间很看重的应用的特殊策略，并且会抢先于使用 `SCHED_OTHER` 策略的进程而执行。一个 `SCHED_FIFO` 进程只能被更高优先级的进程抢先，但一个 `SCHED_RR` 进程必要时可以和其他同级进程共享时间。这些调用在出错时都返回-1（此时还要检测 `errno` 变量）。`sched_setscheduler` 执行成功时返回 0 而 `sched_getscheduler` 执行成功时返回一个非负值。系统调用 `sched_get_priority_max` 和 `sched_get_priority_min` 分别返回对于 `policy` 所指定的策略来说最大和最小的优先级值。`SCHED_OTHER` 进程的静态优先级总是 0；使用 `nice` 或 `setpriority` 可以设置动态优先级。

系统调用 `nice` 通过向调用进程的动态优先级值增加 `inc`，从而降低其优先级。超级用户可以指定一个负值，这能提高进程的优先级。`nice` 执行成功返回 0，执行失败返回-1（通常，要检查 `errno` 并判断出错的真正原因）。

系统调用 `setpriority` 设置进程（`which = PRIO_PROCESS`）、进程组（`which = PRIO_PGRP`）或用户（`which = PRIO_USER`）的动态优先级。优先级的值设置为 `prio`，它可以取介于-20 到 20 之间的某个值，值越小调度优先级越高。该调用执行成功返回 0，如果出错则返回-1（检查 `errno`）。系统调用 `getpriority` 接受的两个参数和 `setpriority` 的头两个参数一样，并且返回所有匹配进程中的最小值（最高优先级）。如果返回-1，则表明出错或者它确实是实际结果；你必须在使用这个函数之前清空 `errno` 变量，并在执行后检查它来判断是哪一种情况。

13.7 小 结

本章相当详细地介绍了 Linux 的进程模型。本章还展示了如何检查并改变进程的属性，如何使用信号控制进程，以及如何查询并改变进程调度。

第 14 章 线程概述

本章介绍线程编程。集中讨论 POSIX 线程，也称为 `pthread`，既因为 POSIX 线程接口可移植性最好，也因为 Linux 对它的支持最好。在定义了什么是线程以后，本章主要介绍 `pthread` 接口。散布在本章各处的程序示范了讨论的内容。

14.1 什么是线程

线程是在共享内存空间中并发的多道执行路径，它们共享一个进程的资源，如文件描述符和信号处理。在两个普通进程（非线程）间进行切换时，内核准备从一个进程的上下文切换到另一个进程的上下文要花费很大的开销。这里上下文切换的主要任务是保存老进程的 CPU 状态并加载新进程的保存状态，用新进程的内存映像替换老进程的内存映像。线程允许你的进程在几个正在运行的任务之间进行切换，而不必执行前面提到的完整的上下文切换。

14.2 `__clone` 函数调用

Linux 特有的函数调用 `__clone` 是 `fork` 的替代函数，它能够更多地控制父进程和子进程之间共享哪些进程资源。

```
#include <sched.h>
int __clone(int (*fn)(void *fnarg), void *child_stack, int flags,
            void *arg);
```

`__clone` 的目的是为了更容易实现 `pthread` 库，很快你就会学到这个库。但能用它以和 `fork` 非常相似的方式创建新的进程。如果你打算编写多线程的程序，应该使用可移植的 `pthread_create` 函数创建适当的线程。下面会详细彻底地讨论 `__clone` 函数。

`(*fn)(void *fnarg)` 是函数指针，当执行子进程时调用这个函数。`fnarg` 是传递给 `fn` 的参数。`child_stack` 是指向你为子进程分配的堆栈的指针。第三个参数 `flags` 通过把表 14.1 中的多种 `CLONE_*` 标志进行“或”操作来得到。第四个参数 `arg` 被传送给子函数；它的取值和用法完全由你决定，因为它为子函数提供了上下文。也就是说，用它从父线程向任何子线程传递数据。`__clone` 返回创建的子进程的进程 ID。在出现错误的情况下，`__clone` 返回 -1 设置 `errno` 变量，并且不创建子进程。

表 14.1 `__clone` 的 flags 参数的取值

值	描述
<code>CLONE_VM</code>	如果设置, 父进程和子进程运行在同一内存空间 (在可应用的地方共享内存映像)
<code>CLONE_FS</code>	如果设置, 父进程和子进程共享在 <code>root</code> 文件系统、当前工作目录以及 <code>umask</code> 信息
<code>CLONE_FILES</code>	如果设置, 父进程和子进程共享文件描述符
<code>CLONE_SIGHAND</code>	如果设置, 父进程和子进程共享设置在父进程上的信号处理器
<code>CLONE_PID</code>	如果设置, 父进程和子进程有相同的 PID

表 14.1 介绍了如果设置了某个标志, `__clone` 表现出的行为。下面的列表描述了如果不设置某个标志会发生什么:

- 如果不设置 `CLONE_VM`, `__clone` 的行为和 `fork` 调用类似, 此时子进程接受它自己的一份父进程内存空间副本, 反之亦然。
- 如果不设置 `CLONE_FS`, 子进程接受一份父进程文件系统信息的副本, 并且 `chroot`、`chdir` 和 `umask` 调用不会影响父进程的根目录、当前工作目录或 `umask`, 反之亦然。
- 如果不设置 `CLONE_FILE`, 子进程仍会继承其父进程的文件描述符表, 但是对那些文件描述符的 `read`、`write`、`open` 和 `close` 操作都不会影响父进程, 反之亦然。
- 如果不设置 `CLONE_SIGHAND`, 子进程继承一份父进程的信号处理器副本, 但是在子进程中的 `sigaction` 调用不会影响父进程的信号处理器, 反之亦然。
- 如果不设置 `CLONE_PID`, 子进程接受一个惟一的 PID。

如前所述, `__clone` 主要是为了实现 Linux 的 `pthread` 库。即使你需要创建多道执行线索, 而它也确实能够用于用户空间的程序, 但我还是强烈建议你使用 `pthread` 接口, 它是本章余下部分的主题。

14.3 `pthread` 接口

本节详细介绍 `pthread` 接口, 并查看 `pthread` 库定义的函数和数据结构。本节还包含了一些示例程序, 用于演示 `pthread` 库显著的特性和基本的使用方法。我们先看看基本的 `pthread` 接口, 然后介绍一些高级特性。

14.3.1 `pthread` 是什么

`pthread` 是一种标准化模型, 它用于把一个程序分成一组能够同时执行的任务。字母 `p` 源自于定义线程应该怎样操作的 POSIX 标准。从程序员的角度来看, 把 `pthread` 看作是由实现 POSIX 线程标准 (如果你想知道得更确切, POSIX Section 1003.1c) 的 C 函数调用和数据结构、提供接口 (`<pthread.h>`) 的头文件, 以及让调用起作用的库 (`libpthread.o`) 组成的集合会方便些。

注意: 还有其他的线程实现, 如 Mach 线程、NT (在 Windows NT 中的) 线程和 DCE 线程。每种模型都以不同的方法实现了线程共同的基本功能, 要创建一个可

移植的、基于线程的应用程序对于应用程序开发人员来说就是一场恶梦。这种分歧就是制订 POSIX 标准的动机。

14.3.2 何时使用 Pthread

线程可能会派上用场的场合包括：

- 在返回前阻塞的 I/O 任务能够使用一个线程处理 I/O，同时继续执行其他处理。
- 需要响应的任何用户界面（UI）能够使用一个或多个线程进行后台处理（比如 I/O——还记得其关键之处吗？）同时保持对用户输入的响应。
- 在有一个或多个任务受不确定性事件，比如网络通信或一种稀有资源的可获得性影响的场合，能够使用线程处理这些异步事件同时继续执行正常的处理。
- 如果某些程序功能比其他的功能更重要，可以使用线程以保证所有功能都出现，但那些时间密集型的功能具有更高的优先级。

这 4 个例子能够普遍地描述成：检查程序中潜在的并行性，也就是说找出能够同时执行的任务。但是聪明的读者会发现 Linux 的进程模型已经提供了执行多个进程的能力，因而已经可以进行并行或并发编程。可是线程能够让你对多个任务的控制程度更好，使用的系统资源更少，因为一个单一的资源，如全局变量，可以由多个线程共享。而且，在拥有多个 CPU 的系统上，多线程应用会比用多个进程实现的应用执行的速度更快。

下面几小节介绍基本的 pthread 调用，它们可以让你使用线程来工作。

14.3.3 pthread_create 函数

首先使用 pthread_create 创建一个新的线程：

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                  pthread_attr_t *attr,
                  void *(*start_routine)(void *),
                  void *arg);
```

pthread_create 在 thread 中保存新线程的标识符。第二个参数 attr 决定了对线程应用哪种线程属性。使用 pthread_attr_init 调用设置新线程的属性，它将在本章后面的“线程属性”一节进行讨论。（*start_routine）是一个指向新线程中要执行的函数的指针。第四个参数 arg 是一个 void 类型的指针，它是传递给（*start_routine）的参数。如果它有意义，则由用户来定义。pthread_create 执行成功便返回 0 并且在 thread 中保存线程的标识符。执行失败则返回一个非零的出错代码。

14.3.4 pthread_exit 函数

pthread_exit 函数使用函数 pthread_cleanup_push 调用任何用于该线程的清除处理函数，然后终止当前线程的执行，返回 retval。retval 可以由父线程或其他线程通过 pthread_join（随后讨论）来检索。一个线程也可以简单地通过从其初始化函数返回来终止。

```
#include <pthread.h>
void pthread_exit(void *retval);
```

14.3.5 pthread_join 函数

函数 `pthread_join` 用于挂起当前线程，直至 `th` 指定的线程终止运行为止。

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
int pthread_detach(pthread_t th);
```

另一个线程的返回值如果不为 `NULL`，则保存在 `thread_return` 指向的地址中。一个线程所使用的内存资源在对该线程应用 `pthread_join` 调用之前不会被重新分配。因而，对于每个可切入的线程必须调用一次 `pthread_join` 函数。线程必须是可切入的而不是处于被分离的状态，并且其他线程不能对同一线程再应用 `pthread_join` 调用。通过在 `pthread_create` 调用中使用一个适当的 `attr` 参数或者调用 `pthread_detach` 可以让线程处于被分离的状态。

注意这里有一个不足之处。与对普通进程（由 `fork` 或 `exec` 创建的那些进程）可以使用众多的 `wait` 调用的不同，在多线程中似乎没有等待某个线程退出的方法。

程序清单 14.1 显示了迄今为止讨论的一些 `pthread` 调用的用法。使用 `make thrdcreat` 可以编译这个程序。

程序清单 14.1 基本的 Pthread 函数

```
/*
 * thrdcreat.c - Illustrate creating a thread
 */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void task1(int *counter);
void task2(int *counter);
void cleanup(int counter1, int counter2);

int g1 = 0;
int g2 = 0;

int main(int argc, char *argv[])
{
    pthread_t thrd1, thrd2;
    int ret;

    /* Create the first thread */
    ret = pthread_create(&thrd1, NULL, (void *)task1, (void *)&g1);
    if(ret) {
        perror("pthread_create: task1");
        exit(EXIT_FAILURE);
    }

    /* Create the second thread */
    ret = pthread_create(&thrd2, NULL, (void *)task2, (void *)&g2);
    if(ret) {
```

```
        perror("pthread_create: task2");
        exit(EXIT_FAILURE);
    }

    pthread_join(thrd2, NULL);
    pthread_join(thrd1, NULL);

    cleanup(g1, g2);

    exit(EXIT_SUCCESS);
}

void task1(int *counter)
{
    while(*counter < 5) {
        printf("task1 count: %d\n", *counter);
        (*counter)++;
        sleep(1);
    }
}

void task2(int *counter)
{
    while(*counter < 5) {
        printf("task2 count: %d\n", *counter);
        (*counter)++;
    }
}

void cleanup(int counter1, int counter2)
{
    printf("total iterations: %d\n", counter1 + counter2);
}
```

虽不令人激动，但示范运行的输出类似下面的结果：

```
$ ./thrdcreat
task1 count: 0
task2 count: 0
task2 count: 1
task2 count: 2
task2 count: 3
task2 count: 4
task1 count: 1
task1 count: 2
task1 count: 3
task1 count: 4
total iterations: 10
$
```

这个程序相当直观，它显示了使用所讨论的 `pthread` 调用的正确步骤和语法。通过在

task1 中加入语句 `sleep(1)`，你可以看到第一个线程，也就是 task1 启动的线程，是怎样被第二个线程（由 `thrd2` 表示）用对自己的 `pthread_join` 调用所中断的。这样做中断了第一个线程的执行，直到第二个线程执行完毕为止。

`thrdcreat` 还揭示了使用线程时要考虑的一个关键问题：同步。同步是指控制访问潜在地可能被同时访问的资源，如全局变量 `g1` 和 `g2`。`cleanup` 函数能够在 task1 和 task2 正更新这两个变量的上下文中访问它们（正如程序执行的过程所显示的那样，因为 `pthread_join` 调用的缘故这种情况并未发生）。必须存在某种机制把对这类资源的访问串行化，或者说同步化，从而让多个线程不会同时访问它们。同步将在 14.3.12 节中进行更详细地讨论。

14.3.6 pthread_atfork 函数

`pthread_atfork` 登记了 3 个处理函数，它们在创建一个新线程的某些时候被调用。该函数的调用语法为

```
#include <pthread.h>
int pthread_atfork( void (*prepare)(void),
                   void (*parent)(void),
                   void (*child)(void));
```

在父进程中 `prepare` 指向的函数在创建新线程之前被调用。`parent` 函数在父进程中随后被调用。`child` 指向的函数在子进程一创建好后就在子进程中被调用。这 3 个函数的任何一个都可以为 `NULL`。如果某个函数为 `NULL`，则不会调用相应的函数。通过多次调用 `pthread_atfork` 可以登记一组以上的处理函数。这些函数能够用于清除在子进程中复制的互斥锁。它们如果执行成功则返回 0，否则返回出错代码。

注意： `pthread_atfork` 的手册页面指出，用 `fork` 创建新线程。这似乎并不正确，在 Linux 里是使用 `__clone` 实现线程的。

提示： 在即将出台的 POSIX 线程标准中，不再包含 `pthread_atfork` 调用，所以可能最好不要在新程序中使用这个函数。

14.3.7 取消线程

`pthread_cancel` 函数允许当前线程取消 `thread` 指定的另一个线程。

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

一个线程可以使用 `pthread_setcancelstate` 来设置它的取消状态，这个函数有两个参数。参数 `state` 是新状态，而参数 `oldstate` 是变量指针，如果 `oldstate` 不为 `NULL`，则这个变量保存有线程的老状态。如果 `state` 为 `PTHREAD_CANCEL_ENABLE`，则允许请求取消。但是，如果 `state` 为 `PTHREAD_CANCEL_DISABLE`，则忽略取消请求。

函数 `pthread_setcanceltype` 改变一个线程对取消请求的响应方式。如果 `type` 为 `PTHREAD_CANCEL_ASYNCHRONOUS`，线程会被立即取消。如果 `type` 为 `PTHREAD_CANCEL_DEFERRED` 则会延迟取消线程直至达到一个取消点。取消点通过调用 `pthread_testcancel` 来建立，如果有任何正被挂起的取消请求，这个函数就会取消当前的线程。前三个函数如果执行成功返回 0，否则返回一个出错代码。`pthread_testcancel` 什么也不返回。

程序清单 14.2 用到了这些函数中的几个。使用 `make thrdcancel` 编译这个程序。

程序清单 14.2 取消线程

```
/*
 * thrdcancel.c - Illustrate thread cancellation
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void task1(int *counter);
void task2(int *counter);
void cleanup(int counter1, int counter2);

int g1 = 0;
int g2 = 0;

int main(int argc, char *argv[])
{
    pthread_t thrd1, thrd2;
    int ret;

    /* Create the first thread */
    ret = pthread_create(&thrd1, NULL, (void *)task1, (void *)&g1);
    if(ret) {
        perror("pthread_create: task1");
        exit(EXIT_FAILURE);
    }

    /* Create the second thread */
    ret = pthread_create(&thrd2, NULL, (void *)task2, (void *)&g2);
    if(ret) {
        perror("pthread_create: task2");
        exit(EXIT_FAILURE);
    }

    pthread_join(thrd2, NULL);
    pthread_cancel(thrd1); /* Cancel the first thread */
    pthread_join(thrd1);

    cleanup(g1, g2);
}
```

```
    exit(EXIT_SUCCESS);
}

void task1(int *counter)
{
    /* Disable thread cancellation */
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    while(*counter < 5) {
        printf("task1 count: %d\n", *counter);
        (*counter)++;
        sleep(1);
    }
    /* Enable thread cancellation */
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
}

void task2(int *counter)
{
    while(*counter < 5) {
        printf("task2 count: %d\n", *counter);
        (*counter)++;
    }
}

void cleanup(int counter1, int counter2)
{
    printf("total iterations: %d\n", counter1 + counter2);
}
```

thrdcancel 和 **thrdcreat** 非常相似。**thrdcancel** 中的新内容包括在 **main** 函数中调用一次 **pthread_cancel**，以及在 **task1** 中调用两次 **pthread_setcancelstate**。第一个调用禁止取消线程，而第二个调用在程序完成其工作后相应地允许取消线程。该程序的示范运行结果表明 **pthread_cancel** 调用没有发挥作用：

```
$ ./thrdcancel
task1 count: 0
task2 count: 0
task2 count: 1
task2 count: 2
task2 count: 3
task2 count: 4
task1 count: 1
task1 count: 2
task1 count: 3
task1 count: 4
total iterations: 10
$
```

但是，如果你把函数 `task1` 中对 `pthread_setcancelstate` 的两次调用注释掉，则 `pthread_cancel` 调用就能发挥预期的作用：

```
$ ./thrdcancel
task1 count: 0
task2 count: 0
task2 count: 1
task2 count: 2
task2 count: 3
task2 count: 4
task1 count: 1
total iterations: 6
$
```

首先，语句 `pthread_join(thrd2, NULL)`；挂起当前线程直至第二个线程（`task2`）完成为止。接着，语句 `pthread_cancel(thrd1)`；取消第一个线程，在这里导致执行 6 次循环。注意，似乎这里应该是 7 次循环（因为输出中有 `task1 count: 1`）。发生这种情况的原因是 `task` 的计时器变量没有用互斥锁锁定。

14.3.8 pthread cleanup 宏

宏 `pthread_cleanup_push` 宏登记了一个处理函数 `routine`，当调用 `pthread_exit` 终止线程或者线程允许取消请求而同时又到达了一个取消点时，就用 `arg` 指定的空指针参数调用这个处理函数。

```
#include <pthread.h>

void pthread_cleanup_push(void (*routine), (void *), void *arg);
void pthread_cleanup_pop(int execute);
void pthread_cleanup_push_defer_np(void (*routine)(void *), void
    *arg);
void pthread_cleanup_pop_restore_np(int execute);
```

处理函数被压入一个栈中，所以宏 `pthread_cleanup_pop` 取消对最近入栈的清除处理函数的登记。如果 `execute` 的值不为 0，处理函数也会被执行。`pthread_cleanup_push` 和 `pthread_cleanup_pop` 必须从同一函数以及同一层次的程序块嵌套中进行调用。这意味着，如果你在一个 `for` 循环的外层调用 `pthread_cleanup_push`，那么就必须在同一层次调用 `pthread_cleanup_pop`，不能在 `for` 循环的更外层或更内层调用它。而且 POSIX 标准要求每个被执行的入栈操作都必须有相应的弹出操作。这些清除宏的主要作用是为了释放由线程分配的，在线程被取消或在调用 `pthread_exit` 终止线程时应该释放的任何资源。但是，即使 POSIX 标准规定了这样的要求，可 Linux 的 `pthread` 实现并不对这一要求做强行规定，所以完全由你本人来决定是否遵守规定。如果你一时兴起没有弹出已经压入栈的处理函数，无论线程存在或是被取消，互斥锁将永远保留下去，结果妨碍了其他使用这个互斥锁的线程正常执行。

宏 `pthread_cleanup_push_defer_np` 是 Linux 特有的扩展，它调用 `pthread_cleanup_push` 和 `pthread_setcanceltype` 推迟执行取消。宏 `pthread_cleanup_pop_restore_np` 弹出最近由

`pthread_cleanup_push_defer_np` 登记的处理函数并保存前一次取消的类型。但是由于这些都是 Linux 特有的调用，所以不要在准备移植的程序中使用它们。

14.3.9 Pthread 条件

本节讨论的函数用于挂起当前线程直到满足某个条件为止。条件是什么能够接收信号的对象。

```
#include <pthread.h>
pthread_cond_t cond = pthread_cond_initializer;
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t
    *cond_attr);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t
    *mutex, const struct timespec *abstime);
int pthread_cond_destroy(pthread_cond_t *cond);
```

函数 `pthread_cond_init` 初始化一个类型为 `cond_t` 的对象 `cond`。第二个参数 `cond_attr` 在 Linux 下被忽略。对于这个参数来说，大多数 Linux 的 pthread 程序只是简单地把 `PTHREAD_COND_INITIALIZER` 复制给 `cond_attr`。函数 `pthread_cond_destroy` 是 `cond_t` 类型对象的析构器 (destructor)。它的功能非常简单：它只是检查没有线程正在等待条件的情况。

函数 `pthread_cond_signal` 用于重启一个并且是惟一的正在等待条件的线程。函数 `pthread_cond_broadcast` 功能与此类似，但它重启所有正在等待条件的线程。`pthread_cond_signal` 和 `pthread_cond_broadcast` 都接受一个参数 `cond`，它表示条件。

函数 `pthread_cond_wait` 解开 `mutex` 指出的一个互斥锁，然后等待变量 `cond` 上的信号。函数 `pthread_cond_timedwait` 功能类似，但是它只等待 `abstime` 指定的时间，`abstime` 时间采用经典的 UNIX 纪元时间，即从 1970 年 1 月 1 日起至今的秒数。因此 `abstime` 和系统调用 `time` 的返回值兼容。本节讨论的所有 pthread 条件调用都可以有取消点，而且所有的函数都在执行成功时返回 0，而在出错时返回出错代码。

14.3.10 pthread_equal 函数

如果参数 `thread1` 和 `thread2` 引用的实际上是同一个线程，则 `pthread_equal` 函数返回一个非零值；否则它返回 0。

```
#include <pthread.h>
int pthread_equal(pthread_t thread1, pthread_t thread2);
```

使用 `pthread_equal` 函数检测线程的等价性非常重要，因为 pthread 标准把怎样定义 `pthread_t` 类型留给每种 pthread 的实现去决定。因此，如果 `pthread_t` 定义成某种 C 结构，那么就不能采用比如 `if(thread1==thread2)` 这样的形式判断线程的等价性，因为不能用这样

的方式比较两个结构（至少在 C 语言中是这样）。始终应该使用 `pthread_equal` 来判断 `pthread_t` 变量引用的线程是否相同。

14.3.11 线程属性

什么是线程属性？线程属性控制着一个线程在它整个生命周期里的行为。表 14.2 列出了线程可能的属性（默认值用星号标出）。高优先级的实时进程可能希望用 `mlock` 锁住它们在内存中的页。`mlock` 还可以由对安全高度敏感的软件用来保护口令字和解锁字不被交换到磁盘上，口令字和解锁字从内存中清除之后还可能永久地保存在磁盘上。

表 14.2 线程属性

属性	值	含义
detachstate	PTHREAD_CREATE_JOINABLE*	可切入的状态
	PTHREAD_CREATE_DETACHED	被分离的状态
schedpolicy	SCHED_OTHER*	正常，非实时
	SCHED_RR	实时，循环（round-robin）
	SCHED_FIFO	实时，先入先出
schedparam	与策略有关	
inheritsched	PTHREAD_EXPLICIT_SCHED*	由 schedpolicy 和 schedparam 设置，从父进程继承
	PTHREAD_INHERIT_SCHED	
scope	PTHREAD_SCOPE_SYSTEM*	一个线程一个系统时间片，线程共享系
	PTHREAD_SCOPE_PROCESS	统时间片（Linux 下不支持）

下面列出的线程属性函数控制着线程属性对象，所以调用它们能够修改线程的默认行为。但要注意，这些调用不能控制和线程直接相关联的属性。产生的属性对象通常要传递给 `pthread_create` 函数。函数 `pthread_attr_init` 清除产生的属性。用户必须在调用这些函数之前为 `attr` 对象分配空间。这些函数能够分配额外的空间，以后再被 `thread_attr_destroy` 释放，但是 Linux 的 `pthread` 实现不在此之列。

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int
    detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int
    *detachstate);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int
    *policy);
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct
    sched_param *param);
int pthread_attr_getschedparam(const pthread_attr_t *attr, struct
    sched_param *param);
int pthread_attr_setinheritsched(pthread_attr_t *attr, int
```

```

inherit);
int pthread_attr_getinheritsched(const pthread_attr_t *attr, int
    *inherit);
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(const pthread_attr_t *attr, int scope);
int pthread_setschedparam(pthread_t target_thread, int policy, const
    struct sched_param *param);
int pthread_setschedparam(pthread_t target_thread, int *policy,
    struct sched_param *param);

```

`pthread_setschedparam` 和 `pthread_getschedparam` 函数分别用来设置和取得与一个运行中的线程相关联的调度策略和参数。`target_thread` 指出要控制的线程, `policy` 是要设置的策略, 它可以取 `SCHED_RR`、`SCHED_FIFO` 或 `SCHED_OTHER` 中的一个值。`SCHED_RR` 和 `SCHED_FIFO` 为目标线程设置实时调度, 而 `SCHED_OTHER` 则使用正常的调度。如果 `policy` 是 `SCHED_RR` 或 `SCHED_FIFO` 中的一个, 那么 `param` 必须为非空并指出线程的优先级。

剩下的函数都只有一个指向要操作的属性对象 `attr` 的指针作为参数, 从它们的名称上就能明显看出它们的作用不是设置就是检索特定属性的值。所有属性控制函数在执行成功时返回 0。在发生错误的情况下, 这些函数返回出错代码而不是设置 `errno` 变量。

14.3.12 互斥

互斥 (mutex) 是相互排斥 (mutual exclusion) 的缩写, 它是一种锁或者信号灯。和信号灯以及文件锁类似, 互斥也用来保护由多个线程共享的数据和结构不被同时修改。一个互斥锁只有两种状态: 加锁 (locked) 和解锁 (unlocked)。但在 `pthread` 的上下文中, 加锁和解锁既有它们正常的含义又有和各自正常含义略微不同的其他一些含义。加锁的互斥不但让其他线程访问, 而且互斥也归上锁进程所有。类似地, 任何线程都能访问解锁互斥, 但它却不归任何线程所有。

正如你所期望的那样, 任何时刻只能有一个线程掌握某个互斥上的锁。一个线程试图在一个已经加锁的互斥上再加锁, 就好像这个互斥归该线程所有一样, 但是这个线程会被挂起, 直到加锁的线程释放掉互斥上的锁为止。与互斥相关的函数如下:

```

#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const
    pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

```

函数 `pthread_mutex_init` 创建指针 `mutex` 指向的互斥, 并且用 `mutexattr` 指定的属性初始化这个互斥。Linux `pthread` 实现把互斥可能的属性限制在表 14.3 列出的属性中。

表 14.3 Linux pthread 互斥属性

属性	描述
PTHREAD_MUTEX_INITIALIZER	创建一个快速互斥
PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP	创建一个递归互斥
PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP	创建一个检错互斥

这三种类型互斥的区别在于当你试图在一个已经加锁的互斥上再加锁（采用随后介绍的 `pthread_mutex_lock`）时它们的行为不同。快速互斥的行为前面已经介绍过：调用线程被阻塞直至拥有互斥的线程解锁为止。检错互斥不阻塞调用线程，但是立即返回一个出错代码 `EDEADLK`。递归互斥成功返回并且增加调用线程在互斥上加锁的次数。在这种情况下，在调用线程能够对互斥解锁之前必须调用同样次数的 `pthread_mutex_unlock`（参见下面的介绍）。

除了使用 `mutexattr` 参数，你可以使用下面的初始化过程静态创建 `pthread_mutex_t` 变量。

```
#include<pthread.h>
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
pthread_mutex_t errchkmutex =
    PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

第一个例子创建了静态初始化的快速互斥，第二个创建了静态初始化的递归互斥，而第三个创建了静态初始化的检错互斥。

`pthread_mutex_destroy` 消除 `mutex` 引用的互斥并且释放互斥占有的任何资源。函数 `pthread_mutex_lock` 和 `pthread_mutex_unlock` 用于分别对互斥 `mutex` 加锁和解锁。正如你所期望的那样，调用 `pthread_mutex_unlock` 的线程必须拥有这个互斥。如果它不拥有互斥，则返回出错代码 `EPERM`。`pthread_mutex_init` 从不会调用失败，但是其他调用 `pthread_mutex_destroy`、`pthread_mutex_lock` 和 `pthread_mutex_unlock` 在执行成功时返回 0，而发生错误时返回一个非零的出错代码。

函数 `pthread_mutex_trylock` 和 `pthread_mutex_lock` 类似，但它不在 `mutex` 已经加锁时阻塞（挂起）调用线程。而是立即返回出错代码 `EBUSY`。因此，如果调用线程没有被阻塞，比如它正在执行时间密集型 I/O 时，那么应该在给一个互斥加锁前调用 `pthread_mutex_trylock` 试验一下。`pthread_mutex_trylock` 执行成功时返回 0，否则返回一个出错代码。

程序清单 14.3 互斥的基本使用

```
/*
 * mutex.c - Using mutexes
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
```



```
#include <errno.h>

#define INDEX 10000000

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
long int ticks;
time_t end_time;

/* An "index" thread to increment a counter */
void idx_th(void *arg);

/* A "monitor" thread to check the counter's value */
void mon_th(void *arg);

int main (int argc, char *argv[])
{
    pthread_t idx_th_id;
    pthread_t mon_th_id;
    int ret;

    end_time = time(NULL) + 30; /* 30 second runtime*/

    /* Create two threads */
    ret = pthread_create(&idx_th_id, NULL, (void *)idx_th, NULL);
    if (ret != 0) {
        perror("pthread_create: idx_th");
        exit(EXIT_FAILURE);
    }
    ret = pthread_create(&mon_th_id, NULL, (void *)mon_th, NULL);
    if (ret != 0) {
        perror("pthread_create: mon_th");
        exit(EXIT_FAILURE);
    }

    pthread_join (idx_th_id, NULL);
    pthread_join (mon_th_id, NULL);

    exit(EXIT_SUCCESS);
}

void idx_th(void *arg)
{
    long l;

    while(time(NULL) < end_time) {
        /* Lock the mutex */
        if(pthread_mutex_lock(&mutex) != 0) {
            perror("pthread_mutex_lock");
            exit(EXIT_FAILURE);
        }

        /* Increment the counter */
        for(l = 0l; l < INDEX; ++l)
```

```

        ++ticks;

        /* Now we're done, so unlock the mutex */
        if(pthread_mutex_unlock (&mutex) != 0) {
            perror("pthread_mutex_unlock");
            exit(EXIT_FAILURE);
        }
        sleep(1);
    }
}

void mon_th(void *arg)
{
    int nlock = 0;
    int ret;

    while(time(NULL) < end_time) {
        /* Wake up every two seconds */
        sleep(3);
        /* Try to lock the mutex */
        ret = pthread_mutex_trylock(&mutex);
        if(ret != EBUSY) {
            if(ret != 0) {
                perror("pthread_mutex_trylock");
                exit(EXIT_FAILURE);
            }
            printf ("mon_th: got lock at %ld ticks\n", ticks);
            if(pthread_mutex_unlock(&mutex) != 0) {
                perror("pthread_mutex_unlock");
                exit(EXIT_FAILURE);
            }
        } else {
            /* Number of times mutex was locked */
            nlock++;
        }
    }
    printf ("mon_th missed lock %d times\n", nlock);
}

```

mutex.c 有两个线程：一个线程 idx_th 执行计时器累加，而另一个线程 mon_th 监视计时器的值。当程序启动时，idx_th 给名为 mutex 互斥加锁，然后用 30 秒对名为 ticks 的计时器进行累加。当 ticks 达到宏 INDEX 定义的值 10 000 000 时，idx_th 就对互斥解锁。休眠一秒钟之后，idx_th 重复上述过程。同时，在同样的 30 秒内，mon_th 每 2 秒醒来一次，并使用 pthread_mutex_trylock 给同一个互斥加锁。如果加锁成功，它就输出一条消息，指出它加锁时 ticks 计时器的值，然后再对互斥解锁。否则，它就对 nlock 变量加 1，这个变量保存了 mon_th 加锁失败的次数。30 秒的时间限度结束后，mon_th 输出它加锁失败的次数并退出。示范运行的结果如下：

```
$ ./mutex
mon_th: got lock at 20000000 ticks
mon_th: got lock at 40000000 ticks
mon_th: got lock at 50000000 ticks
mon_th: got lock at 70000000 ticks
mon_th: got lock at 90000000 ticks
mon_th: got lock at 100000000 ticks
mon_th: got lock at 120000000 ticks
mon_th: got lock at 150000000 ticks
mon_th: got lock at 170000000 ticks
mon_th: got lock at 200000000 ticks
mon_th: got lock at 220000000 ticks
mon_th: got lock at 250000000 ticks
mon_th missed lock 3 times
$
```

注意，ticks 的数量和程序消耗的时间没有内在联系。ticks 的计数值和 mon_th 不能对互斥加锁的次数值会不断变化，这取决于你的处理器速度和系统整体的繁忙程度。例如，在同时运行 SETI@home 程序和使用 updatedb 程序更新本地数据库的系统上再次示范运行这个程序，输出结果如下：

```
$ ./mutex
mon_th: got lock at 20000000 ticks
mon_th: got lock at 30000000 ticks
mon_th: got lock at 50000000 ticks
mon_th: got lock at 80000000 ticks
mon_th: got lock at 100000000 ticks
mon_th: got lock at 130000000 ticks
mon_th: got lock at 150000000 ticks
mon_th: got lock at 160000000 ticks
mon_th: got lock at 180000000 ticks
mon_th: got lock at 210000000 ticks
mon_th: got lock at 230000000 ticks
mon_th: got lock at 240000000 ticks
mon_th missed lock 3 times
$
```

14.4 小 结

本章介绍了庞大而复杂的线程编程话题。通过使用 Linux 的 pthread 实现，它向你展示了创建、破坏以及取消线程的基本接口。本章还通过 pthread_join 调用和互斥解释了线程同步的基本概念。但是实际上本章只触及了这方面的肤浅知识，因为 pthread 编程和线程编程一般说来需要用一本书的篇幅来讲述。

第 15 章 访问系统信息

过去，类似 `ps` 和 `uptime` 这样的 UNIX 程序必须通过直接访问内核的数据结构来检索系统信息。这样做需要掌握内核内部的知识，而且要注意不要在访问那些数据结构的时候修改它们的值。程序必须设置 `setuid` 位为超级用户后才可以访问内核的数据结构；这意味着如果程序编写不当很容易带来安全隐患。这些程序还必须随内核改变而频繁地进行重新编写，因为内核中数据结构的位置和布局都改变了。

某些比较现代的操作系统实现了一个 `/proc` 文件系统，可以通过读取它所包含的特殊文件来访问系统的状态信息。这些文件通常都是纯文本文件，既可以用手工也可以用 UNIX 标准的文件处理工具，如 `cat`、`grep`、`more` 等查看它们。

`/proc` 下的一些条目可以写入信息，这提供了在运行期改变内核参数的手段。例如，一个标准的 2.2.16 版内核分配了 4096 个文件句柄，这对于一些复杂的应用程序，如 `squid` 和 `ircd` 来说太少了。你可以通过下面的命令来使可用的文件句柄数增加一倍：

```
echo 8192 > /proc/sys/fs/file-max
```

Linux 通过它的 `/proc` 文件系统提供了比其他许多系统更多的信息。通常，其他 UNIX 系统只提供当前处于活动状态的进程的信息。即便在活动进程方面，Linux 的 `/proc` 文件系统也提供了比其他大多数操作系统更多的信息。

任何内核模块都能创建和更新 `/proc` 文件系统里的项。在 `/usr/src/linux/fs/proc` 下能够找到实现 `/proc` 文件系统和它的一些项的代码。部分 `/proc` 名字空间由内核函数 `proc_register` 进行登记。

本章描述的 `/proc` 文件系统基于一个运行着 2.2.16 版内核的系统。如果你拥有的系统有不同版本的内核，或者由安装的驱动程序在 `/proc` 下创建了新项，那么你面对的可能是一个稍有不同文件集合。因此，对于某些项来说，你可能要去本书以外的地方寻找所需的信息。

在 UNIX 手册第 5 节中有关 `proc` 的手册页面 (`man 5 proc`) 里有对这些文件更详细的介绍。另一种获得 `/proc` 文件系统信息的来源是内核文档，特别是 `/usr/src/linux/Documentation/proc.txt` 和位于 `/usr/src/linux/Documentation/sysctl/` 目录下的文档。`proc` 中一些更模糊的项，在 `proc` 的手册页面中没有提到，但这些文档却提供了有关这些模糊项的多种设置的丰富信息。这里列举的许多文件都没有在标准文档中提及，所以我不得不依靠自己的知识、内核资源、多种工具的输出的对比，并且偶尔还要猜测一番才最后得到这些描述。

首先要提醒你注意，本章并没有对通过 `/proc` 文件系统所能得到的信息进行细致入微的介绍。本章的内容主要是基于对这些文件的内容、某些使用这些文件的特殊程序的输出、内核源代码以及某些工具程序的源代码进行考察得到的。你能够，并且应该自己完成其中的多数工作，但要把精力集中在当前你的应用程序所需的那些特殊信息上。

本章的主要目的就是让读者知道从系统中能得到什么信息以及从哪里得到。如果你本

来就希望获取一些很特殊的信息，那么也就有了考察/proc 文件系统中特定部分的动力，同时自己编写的代码将成为印证自己想法的手段。或许有人会决定为/proc 文件系统编写一本书或者有一群人要把这一工作作为 Linux 建档计划的一部分来实施；在那之前，程序员们应该打算自力更生，自己研究这部分内容。

打个比方来作总结：我会把读者引入图书馆，并告诉读者如何使用分类卡片。至于仔细阅读的工作则是读者自己的事情。

15.1 进 程 信 息

系统中任何时刻正在运行的每个用户级进程在/proc 下都有一个目录；目录的名称就是进程号的十进制表示。另外，/proc/self 是链接到访问/proc 目录的进程自己目录的符号链接（这个链接对每个进程看起来都不同）。这些目录下存放着许多文件。

在下面各小节中，\$pid 可以为任何感兴趣的进程的进程 ID 所替代。这些文件中的大多数都可以用 UNIX 标准的文本处理工具查看，比如 more、cat、strings 或 grep。

提示： strings 的 -f 选项能输出每个被处理文件的文件名，这个选项和通配符联用会很方便。

注意： 源文件/usr/src/linux/fs/proc/array.c 中的大多数例程实际上是用来产生每个进程项的/proc 输出的。

15.1.1 cmdline 文件

文件/proc/\$pid/cmdline 内容只有一行，它是进程的命令行，包括程序的名称和所有的参数。僵进程的命令行可能是空行，另外如果进程被交换出内存，可能就得不到程序的参数。

提示： 你可以通过发出下面的命令来快速查看系统中正在运行什么进程：

```
strings -f /proc/[0-9]*/cmdline
```

警告： 因为 cmdline 项具有组可读权限，所以所有的用户都能访问其中的信息。因此，你要注意自己在命令行上提供信息的内容。例如，绝对不要包含口令字或别的你不希望其他用户看到的类似信息。当你键入命令时一定要记住这一点。不要让用户在命令行上提供有权限的信息。当程序开始运行后提示用户输入这些信息的方法要好得多。

15.1.2 environ 文件

文件/proc/\$pid/environ 记录了进程的环境信息。单个的环境字符串之间由空字节分隔，以文件结束标志作为环境结束的标志。

提示： 用工具 strings 显示 environ 项的内容要比用工具 cat 显示的可读性更好。

15.1.3 fd 目录

目录 `/proc/$pid/fd` 为每个打开的文件描述符提供了一个入口，它是到实际文件的索引节点 (inode) 的符号链接。一个索引节点包含了一个文件的信息。每个索引节点都保存有索引节点所在的设备信息、加锁信息、文件的模式和类型、到该文件的链接、文件所有者和所有组 ID、文件字节数以及文件在设备上的块数。

打开文件描述符入口 (如 `/proc/self/fd/1`)，也就打开了文件本身；如果该文件是个终端或者其他特殊设备，这样做可能会因为窃取数据而干扰进程本身的执行。

在 `fd` 目录下还有一些指向特殊文件的项。0 指向标准输入文件，1 指向标准输出文件，而 2 指向标准出错文件。

提示： 有些程序要么不接受来自标准输入文件的输入，要么不向标准输出文件发送输出。通过一些技巧和 `/proc` 文件系统，你可以欺骗这些程序，让它们成为过滤器。你可以把 `/proc/self/fd/0` 设置为输入文件而把 `/proc/self/fd/1` 设置为输出文件，就能实现上述效果。

你可以使用系统调用 `fstat` 或 `lstat` 取得进程正在处理的文件的信息。你也可以使用 `stat` 命令得到这些信息。文件显示出的权限只定义了所有者本人的权限，而所有者读权限位和所有者写权限位则表明了文件打开的模式。

在许多 UNIX 系统上，无论有没有 `/proc` 文件系统，都有一个叫作 `lsof` 的工具程序。在 Red Hat 系统上，这个工具的安装位置是 `/usr/sbin/lsof`。当你想要知道一个进程到底在干什么，想知道为什么某个文件系统因为忙而不能卸掉，或者想调查可疑的活动，那么这个工具就会发挥作用。`fuser` 命令和 `lsof` 类似，但它只是检查特定文件。如果你用了老的发布版本或者没有完全安装系统，那么有可能找不到这两个工具。

提示： 程序员尤其应该考虑完整安装 Linux 发布光盘上的每一个软件。

15.1.4 mem 文件

可以通过 `/proc/self/mem` 文件来访问特定进程的内存映像。不能使用类似 `strings` 这样的普通工具，但如果你拥有足够的权限可以调用 `mmap` 实现上述目的。

15.1.5 stat

文件 `/proc/$pid/stat` 包含有通常应该由 `ps` 显示的有关某个进程的大多数信息。表 15.1 列出了这些信息域、以及它们的位置、域在 `ps` 程序中的名称和对域的描述。累加值包括当前进程和任何已经结束的子进程 (使用 `wait` 调用族) 的值。许多与时间相关的域是以“瞬间 (jiffy, 1/100 秒)”为单位计算的。

表 15.1 stat 域

域编号	名称	描述
1	pid	进程号
2	cmd	被括号括起来的命令行基本名 (如果命令行为空，则由 <code>ps</code> 使用)

(续表)

域编号	名称	描述
3	state	R=可运行, S=睡眠, D=不能中断的睡眠, T=被跟踪或停止, Z=僵进程, W=不驻留, N=谦让度
4	ppid	父进程号
5	pgrp	进程组号
6	session	进程会话号
7	tty	起控制作用的 TTY, 这是一个由主设备号和次设备号联合构成的 16 比特位的值。如果没有控制终端 TTY, 则返回一个 0 值。
8	tpgid	控制终端 TTY 的进程号
9	flags	进程标志
10	min_fit	次要页面缺失
11	cmin_fit	次要页面缺失 (累计)
12	maj_fit	主要页面缺失
13	cmaj_fit	主要页面缺失 (累计)
14	utime	用户时间
15	stime	系统时间
16	cutime	用户时间 (累计)
17	cstime	系统时间 (累计)
18	priority	静态调度优先级
19	nice	普通调度算法的“谦让度”级别 (动态优先级)
20	timeout	以“瞬间”为单位的超时值。这个值被忽略, 总是返回 0。
21	it_real_value	以“瞬间”为单位的下次超时到期的时间间隔
22	start_time	进程启动时间
23	vsize	以字节计算的 VM (虚存) 大小 (全部)
24	rss	驻留集大小
25	rss_rlim	RSS rlimt
26	start_code	代码段起始值
27	end_code	代码段结束值
28	start_stack	堆栈段起始值
29	kstk_esp	当前堆栈帧
30	kstk_eip	当前堆栈帧
31	signal	挂起信号
32	blocked	被阻塞信号
33	sigignore	被忽略信号
34	sigcatch	捕获信号——带有处理函数的信号
35	wchan	进程睡眠的内核函数名
36	nswap	页交换
37	cnsnap	页交换 (累计)
38	exit_signal	接收退出信号
39	processor	执行 SMP (对称多处理机) 任务的处理器

15.1.6 status 文件

文件 `/proc/$pid/status` 比 `/proc/$pid/stat` 包含的信息少, 但具有更好的可读格式。这个文件包含进程的名称、状态、进程号、父进程号、用户 ID 和组 ID (包括真实的、有效的以及保存的)、虚存统计以及信号掩码。

15.1.7 cwd 符号链接

符号链接 `/proc/$pid/cwd` 指向进程的当前工作目录的索引节点。

15.1.8 exe 符号链接

`/proc/$pid/exe` 是到正在被执行的文件的符号链接。它通常指向一个二进制文件。它还可以指向一个正被执行的脚本文件或者可执行程序正在处理的文件, 这取决于脚本的内容。执行 `/proc/$pid/exe` 会启动该程序的一个新进程。

15.1.9 maps 文件

文件 `/proc/$pid/maps` 记录了有关进程的内存映射区的信息。它包括地址范围、权限、偏移量 (在映射文件中) 以及主次设备号和映射文件的索引节点。

15.1.10 root 符号链接

符号链接 `/proc/$pid/root` 链接到进程的根目录 (由系统调用 `chroot` 设置)。

15.1.11 statm 文件

特殊的文件 `/proc/$pid/statm` 列出了一个进程对内存的使用情况。`array.c` 中用到的变量名依次为 `size`、`resident`、`share`、`trs`、`lrs`、`drs` 和 `dt`。它们分别给出了内存总大小 (包含了代码段、数据段和堆栈段)、驻留集大小、共享页面数、文本页面数、堆栈页面数和脏页面数。注意, 在讨论内存使用情况时“文本”一词经常意味着是可执行代码。`top` 程序使用这些信息中的一部分用于它的输出。

15.2 一般系统信息

`/proc` 文件系统下的多种文件提供的系统信息不是针对某个特定进程的, 而是能够在整个系统范围的上下文中使用。可以使用的文件随系统配置的变化而变化。命令 `procinfo` 能够显示基于其中某些文件的多种系统信息。

15.2.1 /proc/cmdline 文件

这个文件给出了内核启动的命令行。它和用于进程的 `cmdline` 项非常类似。例如,

```
auto BOOT_IMAGE=linux ro root=805 BOOT_FILE=
/boot/vmlinuz-2.2.16-17
```


15.2.2 /proc/cpuinfo 文件

这个文件提供了有关系统 CPU 的多种信息。这些信息是从内核里对 CPU 的测试代码中得到的。文件列出了 CPU 的普通型号 (386、486、586 等等), 以及能得到的更多特定信息 (制造商、型号和版本)。文件还包含了以 bogomips 表示的处理器速度, 而且如果检测到 CPU 的多种特性或者 bug, 文件还会包含相应的标志。这个文件的格式为: 文件由多行构成, 每行包括一个域名称、一个冒号和一个值。

15.2.3 /proc/devices 文件

这个文件列出字符和块设备的主设备号, 以及分配到这些设备号的设备名称。

15.2.4 /proc/dma 文件

这个文件列出由驱动程序保留的 DMA 通道和保留它们的驱动程序名称。cascade 项供用于把次 DMA 控制器从主控制器分出的 DMA 行所使用; 这一行不能用于其他用途。

15.2.5 /proc/file systems 文件

这个文件列出可供使用的文件系统类型, 一种类型一行。虽然它们通常是编入内核的文件系统类型, 但该文件还可以包含可加载的内核模块加入的其他文件系统类型。

15.2.6 /proc/interrupts 文件

这个文件的每一行都有一个保留的中断。每行中的域有: 中断号、本行中断的发生次数、可能带有一个加号的域 (SA_INTERRUPT 标志设置), 以及登记这个中断的驱动程序的名字。在 /usr/src/linux/arch/i386/kernel/irq.c (在 Intel 平台上) 中的 get_irq_list 函数产生这些数据。

可以在安装新硬件之前, 像查看 /proc/dma 和 /proc/ioports 一样用 cat 命令手工查看手头的这个文件。这几个文件列出了当前投入使用的资源 (但是不包括那些没有加载驱动程序的硬件所使用的资源)。

15.2.7 /proc/ioports 文件

这个文件列出了诸如磁盘驱动器、以太网卡和声卡设备等多种设备驱动程序登记的许多 I/O 端口范围。

15.2.8 /proc/kcore 文件

这个文件是系统的物理内存以 core 文件格式保存的文件。例如, GDB 能用它考察内核的数据结构。它不是纯文本, 而是 /proc 目录下为数不多的几个二进制格式的项之一。

15.2.9 /proc/kmsg 文件

这个文件用于检索用 printk 生成的内核消息。任何时刻只能有一个具有超级用户权限的进程可以读取这个文件。也可以用系统调用 syslog (不要把它和库函数 syslog 混淆) 检索这些消息。通常使用工具 dmesg 或守护进程 klogd 检索这些消息。

15.2.10 /proc/ksyms 文件

这个文件列出了已经登记的内核符号；这些符号给出了变量或函数的地址。每行给出一个符号的地址、符号名称以及登记这个符号的模块。程序 `ksyms`、`insmod` 和 `kmod` 使用这个文件。它还列出了正在运行的任务数、总任务数和最后分配的 PID。

15.2.11 /proc/loadavg 文件

这个文件给出以几个不同的时间间隔计算的系统平均负载，这就如同 `uptime` 命令显示的结果那样。前三个数字是平均负载。这是通过计算过去 1 分钟、5 分钟、15 分钟里运行队列中的平均任务数得到的。随后是正在运行的任务数和总任务数。最后是上次使用的进程号。

15.2.12 /proc/locks 文件

这个文件包含在打开的文件上的加锁信息。它是由 `/usr/src/linux/fs/locks.c` 中的 `get_locks_status` 函数产生的。文件中的每一行描述了特定文件和文档上的加锁信息以及对文件施加的锁的类型。内核也可以需要时对文件施加强制性锁。这种信息出现在 `/proc/locks` 文件中。在 `/usr/src/linux/Documentation` 子目录下的文档 `locks.txt` 和 `mandatory.txt` 讨论了 Linux 下的文件加锁。

15.2.13 /proc/mdstat 文件

这个文件包含了由 `md` 设备驱动程序控制的 RAID 设备信息。

15.2.14 /proc/meminfo 文件

这个文件给出了内存状态的信息。它显示出系统中空闲内存、已用物理内存和交换内存的总量。它还显示出内核使用的共享内存和缓冲区总量。这些信息的格式和 `free` 命令显示的结果类似。

15.2.15 /proc/misc 文件

这个文件报告用内核函数 `misc_register` 登记的设备驱动程序。

15.2.16 /proc/modules 文件

这个文件给出可加载内核模块的信息。`lsmod` 程序用这些信息显示有关模块的名称、大小、使用数目方面的信息。

15.2.17 /proc/mounts 文件

这个文件以 `/etc/mtab` 文件的格式给出当前系统所安装的文件系统信息。这个文件也能反映出任何手工安装从而在 `/etc/mtab` 文件中没有包含的文件系统。

15.2.18 /proc/pci 文件

这个文件给出 PCI 设备的信息。用它可以方便地诊断 PCI 问题。你可以从这个文件中

检索到的信息包括诸如 IDE 接口或 USB 控制器这样的设备，总线、设备和功能编号，设备延迟以及 IRQ 编号。

15.2.19 /proc/rtc 文件

这个文件给出硬件实时时钟的信息，包括当前日期和时间、闹钟设置、电池状态和多种支持特性。命令/sbin/hwclock 通常用于控制实时时钟。

15.2.20 /proc/stat 文件

这个文件包含的信息有 CPU 利用率、磁盘、内存页、内存对换、全部中断、接触开关以及上次自举时间（自 1970 年 1 月 1 日起的秒数）。

15.2.21 /proc/uptime 文件

这个文件给出自从上次系统自举以来的秒数，以及其中有多少秒处于空闲。这主要供 uptime 程序使用。比较这两个数字能够告诉你长期来看 CPU 周期浪费的比例。

15.2.22 /proc/version 文件

这个文件只有一行内容，说明正在运行的内核版本。其内容举例如下：

```
Linux version 2.2.16-17 (root@porky.devel.redhat.com)
(gcc version egcs-2.91.66 19990314/Linux (egcs-1.1.2 release))
#1 Wed Jul 26 16:16:19 EDT 2000
```

/proc/version 的输出只有一个文本字符串，可以用标准的编程方法进行分析获得所需的系统信息。

15.2.23 /proc/net 子目录

/proc/net 子目录下的文件描述或修改了联网代码的行为。可以通过使用 arp、netstat、route 和 ipfwadm 命令设置或查询这些特殊文件中的许多文件。表 15.2 列出了多种不同的文件和它们的功能。

表 15.2 /proc/net/文件

文件	描述
arp	转储每个网络接口的 arp 表中 dev 包的统计
dev	来自网络设备的统计
dev_mcast	列出二层（数据链路层）多播组
dev_stat	网络设备状态
igmp	加入的 IGMP 多播组
ip_masq	包含 IP 伪装表的目录
ip_masquerade	IP 伪装连接的信息
netlink	netlink 套接口的信息

(续表)

文件	描述
netstat	网络流量的多种统计。第一行是信息头，带有每个变量的名称。接下来的一行保存相应变量的值
raw	原始套接口的套接口表
route	静态路由表
rpc	包含 RPC 信息的目录
rt_cache	路由缓冲
snmp	snmp agent 的 ip/icmp/tcp/udp 协议统计；各行交替给出字段名和值
sockstat	列出使用的 tcp/udp/raw/pac/syn_cookies 的数量
tcp	TCP 连接的套接口表
tr_rif	令牌环 RIF 路由表
udp	UDP 连接的套接口表
unix	UNIX 域套接口的套接口表
wireless	无线接口数据

15.2.24 /proc/scsi 子目录

/proc/scsi 目录下包含一个列出了所有检测到的 SCSI 设备的文件，并且为每种控制器驱动程序提供一个目录，在这个目录下又为已安装的此种控制器的每个实例提供一个子目录。表 15.3 列出了 /proc/scsi 下的文件和子目录。

表 15.3 /proc/scsi/文件

文件	描述
/proc/scsi/\$driver/\$n	每个控制器一个文件，\$driver 是 SCSI 控制器驱动程序的名字，\$n 是一个编号
/proc/scsi/scsi	列出所有检测到的 SCSI 设备；可以写入特殊的命令以探测特定的目标地址，比如 scsi singledevice 1 0 5 0 在通道 0 上探测设备号 5

15.2.25 /proc/sys 子目录

在 /proc/sys 目录下有许多子目录。在 /proc/sys 目录树下的许多项都可以用来调整系统的性能。表 15.4 包含了对该目录下多种文件的描述。这个目录包含的信息太多，以至于无法在这里介绍所有的项。某些项的含义相当模糊也不常用。我只试着介绍表中较为有趣的一些项。请参考前面讨论过的内核文档了解更多的信息。

表 15.4 /proc/sys/文件

文件	描述
dev	包含多种设备驱动程序的信息。这些信息随系统的不同而不同，取决于使用的设备。在笔者的系统上有一个用于 CD-ROM 的子目录，其中包含了多种用于该设备的驱动程序的参数
fs/binfmt_misc	这个目录处理内核对各种不同二进制文件格式的支持

(续表)

文件	描述
fs/dentry-state	目录项缓冲的状态
fs/dquot-max	被缓冲的磁盘限额项的最大数目
fs/dquot-nr	分配和使用的磁盘限额项的数目
fs/file-max	内核分配的文件句柄的最大数目
fs/file-nr	已分配的文件句柄数目、已使用的文件句柄数目和文件句柄的最大数目
fs/inode-max	索引节点句柄的最大数目
fs/inode-nr	已分配的索引节点数目和空闲的索引节点数目
fs/inode-state	和 fs/inode-nr 中的值相同，后跟 preshrink 值
fs/super-max	超级块句柄的最大数目
fs/super-nr	已分配的超级块数目
kernel/acct	进程记帐控制值
kernel/cap-bound	能力
kernel/ctrl-alt-del	键盘上 Ctrl+Alt+Delete 键的作用
kernel/domainname	域名
kernel/hostname	主机名
kernel/osrelease	内核版本号
kernel/ostype	操作系统名称，即“Linux”
kernel/panic	取得/设置应急故障超时秒数；如果大于 0，则内核在出现应急故障后等待这一秒数然后重启
kernel/printk	内核消息日志级别
kernel/real-root-dev	root 设备的 16 比特位设备号。用公式：主设备号×256+次设备号计算得到这个设备号。
kernel/rtsig-max	系统中显著的 POSIX 实时信号的最大数目
kernel/trsig-nr	当前队列中实时信号的数目
kernel/shmall	共享内存的最大值
kernel/shmmax	能够创建的共享内存区大小的最大值
kernel/sysrq	指出 magic SysRQ 关键字是否启用
kernel/version	编译时间
net/core	通用联网参数
net/core/message_burst	和 message_cost 联用限制写入内核日志的警告消息数目
net/core/message_cost	参见/net/core/message_burst
net/core/netdev_max_backlog	输入端队列中的最大包数
net/core/optmem_max	每个套接口允许的最大的辅助缓冲大小
net/core/rmem_default	套接口读缓冲大小的默认值
net/core/rmem_max	套接口读缓冲大小的最大值
net/core/wmem_default	套接口写缓冲大小的默认值
net/core/wmem_max	套接口写缓冲大小的最大值
net/ipv4	标准 IP 联网参数。这里包含的项太多，在此无法完全说明（我的/net/ipv4 中有 40 多项）。参考/usr/src/linux/Document/proc.txt 获得所有项的列表

(续表)

文件	描述
net/token-ring	此目录包含多种用于令牌环网的参数
net/unix	此目录包含 UNIX 域套接口的参数
proc	此目录为空
sunrpc	此目录包含用于 NFS 的参数
vm/bdflush	磁盘缓冲强行物理写入的参数
vm/buffermem	控制应该为缓冲内存使用多少内存。包含 3 个值: min_percent、borrow_percent 和 max_percent。内核不使用后两项。
vm/freepages	设置/取得 min_freepages(man stm)
vm/kswapd	/usr/include/linux/swapctl.h

15.3 未来内核中/proc 的变化

在撰写本书的时候 Linux 的 2.4 版内核尚未完成。只能获得几种预先发布的版本, 这些版本之间由许多改动。但是, /proc 系统改动不大, 如果你已经看到本书的这个位置, 那么大多数项对你来说应该都很熟悉。

内核开发的邮递列表中有警告说未来版本的内核中/proc 文件系统将会有重大变化。需要对其进行清理, 清除几项。而且有人认为目前/proc 下的许多项根本不属于/proc。

这些变动是否会在 2.4 版或者以后版本的内核中出现还不清楚。但要做好准备, 因为你可能不得不重新组织依赖于/proc 消息的程序。如果可能, 最好使用 API 调用, 比如 libc 中的那些函数, 来取得信息, 因为这样做更能保证所用到信息不会被改变。

如果你正在开发的程序非常依赖于保存在/proc 下的信息, 那么加入某些内核开发的邮递列表或者至少阅读一下涉及内核的文献摘要可能是个不错主意。实际上, 如果你对内核中的任何东西都感兴趣, 就应该阅读这些资源, 因为它们充满了有趣的内容。你可以在 <http://www.linuxcare.com/> 中找到 “Kernel Traffic”, 它是 Linux 内核开发邮递列表的周报。

15.4 小 结

/proc 文件系统包含了大量的有关当前系统状态的信息。通常查看/proc 下文件的内容就能够对这些文件的意思相当清楚了, proc 的手册页面中也有对这些文件的解释文档。把文件和分析这些文件的工具产生的输出进行比较能够更加清晰地了解这些文件。另外, 有兴趣的读者也可以通过考察内核源代码或者考察使用/proc 文件系统取得信息的程序或库的源代码来了解文件中各个字段的含义。

第 16 章 内存管理

从许多方面来看，Linux 系统的内存管理都能和任何现代 PC 操作系统的内存管理相媲美。本章先回顾基本的 C 内存管理，然后解释 Linux 提供的一些附加功能。特别地，你会看到内存映射文件，这是一种执行输入输出的快速方法，还有内存加锁技术，这是一种能够把重要的或者时间密集型数据保存在活动内存中而不被交换到磁盘上的方法。你还会学到如何使用 Electric Fence，它是一种调试内存问题的特殊工具。

注意： 术语“内存锁定”有两个含义。在某些上下文中，它是指避免内存区被同时访问。在另外一些上下文中，它是指把数据保存在内存中，防止把它交换到磁盘或写入磁盘。本章使用“内存锁定”一词表示把数据保存在内存中不让它写入磁盘。

16.1 C 内存管理回顾

C 语言用 `malloc`、`calloc`、`realloc` 和 `free` 函数提供了对动态内存分配的支持。这些函数可以让用户根据需要从操作系统中获取、使用和释放内存。动态内存分配是编制高效率程序的基础。除了能够高效率地使用内存和重要的系统资源外，动态内存管理还能够把程序员从在代码中添加任意的限制中解放出来。有了动态内存管理，用户不再人为地限制数组，比如字符串数组的长度，而是能够请求更多的内存资源从而避免硬编码的限制。下面几节将逐一讨论这些函数。

16.1.1 malloc 函数的使用

`malloc` 函数分配没有被初始化过的内存块。它分配下面原型中 `size` 所指定的字节数的内存，并返回指向新分配的内存的指针，如果执行失败则返回 `NULL`。

```
#include <stdlib.h>
void *malloc(size_t size);
```

通常必须检查 `malloc` 的返回值。下面的代码片段显示了分配内存并检查 `malloc` 返回值的标准调用规范，当然被返回的内存的实际类型可以有所变化：

```
#include <stdlib.h>
char *pmem;
if((pem = malloc(sizeof(char) * 100)) == NULL) {
    /* code here handles the failure */
}
```

不必对 `malloc` 返回的指针做强制类型转换，因为在赋值的时候会自动转换为正确的类

型，但是你会在老的、ANSI 标准之前的代码中看到这种类型转换的用法。你得到的内存块没有进行初始化，所以在正确地初始化之前不要使用它。为防止出现内存泄漏，使用 `malloc` 得到的内存必须再调用 `free` 释放以返回给操作系统。

注意： 一般情况下，你可以将一个 `void` 类型的指针赋值给任何其他类型的指针变量，而不会丢失任何信息，反之亦然。

提示： 如果你遇到了对 `malloc` 调用返回的空指针做强制类型转换的代码，那么就把它改正过来。接着，如果可能就把改动反馈给维护这段代码的人。C 标准允许强制类型转换是为了保持向后兼容性，但是在来来版本的标准中将不再提供这种支持。

16.1.2 `calloc` 函数的使用

`calloc` 函数分配并初始化内存块。它的函数原型如下：

```
#include <stdlib.h>
void *calloc(size_t nmem, size_t size);
```

这个函数的功能和 `malloc` 非常相似，它返回一个指向包含 `nmem` 个元素的数组的指针，数组中每个元素的大小为 `size` 个字节。它和 `malloc` 的不同之处在于，`calloc` 对分配到的内存进行初始化，把每个比特位设置为 0。该函数在执行成功时返回指向内存的指针，在执行失败时返回 `NULL`。

16.1.3 `realloc` 函数的使用

`realloc` 函数能够改变以前分配的内存块的大小。可以使用 `realloc` 调整以前由 `malloc` 或 `calloc` 调用获得的内存的大小。这个函数的原型如下：

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

参数 `ptr` 必须是由 `malloc` 或 `calloc` 返回的指针。参数 `size` 既可以大于原来指针指向的内存块的大小，也可以小于它。增大或减小的操作是原地，也就是在相对于内存块当前地址的地方进行的。如果不能这样做，`realloc` 就把原来的数据复制到新的位置。但是，程序员必须要调整指针以便正确地访问新的内存块。下面几点也适用于 `realloc` 函数：

- `realloc` 不对增加的内存块做初始化。
- `realloc` 如果不能扩大内存块，就返回 `NULL`，而且保持原来的数据不动。
- `realloc` 的第一个参数如果为 `NULL`，则它的作用和 `malloc` 函数一样。
- `realloc` 的第二个参数如果为 0，则释放原来的内存块。

16.1.4 `free` 函数的使用

`free` 函数释放一块内存。这个函数的原型如下：

```
#include <stdlib.h>
void free(void *ptr);
```


参数 `ptr` 必须是先前调用 `malloc` 或 `calloc` 返回的指针。试图访问已经被释放的内存将会导致出错。

内存分配函数从一个称为堆 (heap) 的存储池中获得内存。内存作为一种有限的资源可能会被用尽, 所以用完内存后一定要释放它。同时也要注意出现悬空指针 (dangling pointer)。永远不把分配到的内存释放回操作系统会引起内存泄漏。悬空指针是在内存被释放后遗留下来未初始化的指针。通常悬空指针不算什么问题。只有当你试图访问一个已经释放的指针而又没有重新初始化过它所指向的内存时, 才会引出麻烦, 下面的代码片段说明了这个问题:

```
char *str;
str = malloc(sizeof(char) * 4);
free(str);
strcpy(str, "abc");
```

出毛病啦! 程序运行到最后一行时会告诉你出现了段错误 (SIGSEGV)。

16.1.5 `alloca` 函数的使用

`alloca` 函数分配一块未经初始化的内存。这个函数的原型如下:

```
#include <stdlib.h>
void *alloca(size_t size);
```

迄今为止, 我们介绍的动态内存分配函数 `malloc`、`calloc` 和 `realloc` 都是从堆中获得它们的内存。`alloca` 也与此类似, 但不同之处在于它是从进程的栈而不是堆中获得内存的, 而且当调用 `alloca` 的函数返回时, 已分配的内存会被自动释放。

16.2 内存映像文件

虽然严格地说, 不应该把内存映像文件归入“内存管理”这一标题之下, 这里介绍它是因为它是 Linux 如何管理内存的一个例子。Linux 允许任何进程把一个磁盘文件映像到内存中, 在磁盘文件和它在内存中的映像间创建逐字节的对应关系。

使用内存映像文件有两个主要的优点。首先它可以加速文件 I/O 操作。普通的 I/O 调用, 比如系统调用 `read` 和 `write` 或者库调用 `fputs` 和 `fgets` 通过内核缓冲读出或写入数据。虽然 Linux 具有一种快速而先进的磁盘缓冲算法, 但是最块的磁盘访问也总是要比最慢的内存访问还要慢。在内存映像文件上的 I/O 操作跳过了内核缓冲, 因而速度要快许多。在内存映像文件上执行操作也比较简单, 因为你可以用指针而不是普通的文件操作函数来访问内存映像文件。

使用内存映像文件的另一个优点是可以共享数据。如果多个进程需要访问同样的数据, 这些数据就可以保存在一个内存映像文件中, 所有的进程都能够访问它。作为一种高效的共享内存模型, 内存映像文件能够向任何进程独立地提供数据访问, 并且把内存区的内容保存在一个磁盘文件中。如果你选择这样的方式使用内存映像文件, 还要对内存中的数据

采取一种串行访问 (serializing access) 的方法, 以保证统一的、可预测的读写操作。串行访问是指使用锁或信号灯 (或者某些其他机制) 来避免多个进程同时访问数据。在这种情况下, 使用共享内存可能会更简单一些, 共享内存将在第 17 章“进程间通信”中进行讨论。

Linux 提供了一系列调用管理内存映像。这些函数在 `<sys/mman.h>` 中定义, 包括 `mmap`、`munmap`、`msync`、`mprotect`、`mlock`、`munlock`、`mlockall` 和 `munlockall`。接下来的几节逐一详细讨论这些函数。

16.2.1 mmap 函数的使用

`mmap` 函数把一个磁盘文件映像到内存中。它的原型如下:

```
#include <unistd.h>
#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot, int flags, int fd,
           off_t offset);
```

在文件描述符 `fd` 指定的打开文件中, 从文件起始处偏移 `offset` 的位置开始映像到内存中 `start` 开始的地方。`length` 指定了文件被映像的大小。映像的内存区有保护模式, 其值可以是表 16.1 中列出的值的逻辑“或”。映像的属性由 `flags` 指定, 其值可以是表 16.2 中列出的值的逻辑“或”。如果 `mmap` 执行成功则返回指向内存区的指针。如果执行失败则返回 `-1` 并设置 `errno` 变量。

表 16.1 保护模式值

保护	访问权限
<code>PROT_NONE</code>	不允许访问
<code>PROT_READ</code>	被映像内存区可读
<code>PROT_WRITE</code>	被映像内存区可写
<code>PROT_EXEC</code>	被映像内存区可执行

注意: 在 x86 体系结构中, `PROT_EXEC` 隐含了 `PROT_READ`, 所以 `PROT_EXEC` 和 `PROT_EXEC|PROT_READ` 的作用相同。

表 16.2 标志值

标志	POSIX 兼容性	描述
<code>MAP_FIXED</code>	兼容	如果 <code>start</code> 无效或者正在使用则失败
<code>MAP_PRIVATE</code>	兼容	对映像内存区的写入操作是进程私有的
<code>MAP_SHARED</code>	兼容	对映像内存区的写入也被复制到文件
<code>MAP_ANON (MAP_ANONYMOUS)</code> ¹	不兼容	创建匿名映像, 忽略 <code>fd</code>
<code>MAP_DENYWRITE</code>	不兼容	不允许正常的写文件
<code>MAP_GROWSDOWN</code>	不兼容	映像内存区是向下增长的
<code>MAP_LOCKED</code>	不兼容	把页面锁定在内存中

¹ 译者注: `MAP_ANON` 和 `MAP_ANONYMOUS` 通用。

offset 通常为 0，表明整个文件都被映像到内存中。

映像内存区必须用 `MAP_PRIVATE` 标为私有，或者用 `MAP_SHARED` 标为可共享；其他的值都是可选值。私有映像使得对映像内存区的任何修改都是进程私有的，所以它们不会反映到下面的磁盘文件中，其他进程也不会得到这些修改。另一方面，共享映像使得对映像内存区的任何更新都能够立即让使用同一文件做映像的其他进程看到。为了防止写入下层的磁盘文件，可以指定 `MAP_DENYWRITE`（但要注意这个值不兼容 POSIX 标准，因而也不能移植）。由 `MAP_ANONYMOUS` 创建的匿名映像不包含物理文件，只是简单地分配内存，比如高速 I/O 区域或者自定义的 `malloc` 实现。`MAP_FIXED` 让内核把映像定位在特定的地址。如果这个地址已经在用了或者不能使用这个地址，则 `mmap` 执行失败。如果没有指定 `MAP_FIXED` 而且 `start` 也不能用，内核会尝试把映像放置到内存中的其他地方。`MAP_LOCKED` 允许具有超级用户权限的进程把映像锁定在内存中，而不让它被交换到磁盘上。用户空间程序不能使用 `MAP_LOCKED`，这个安全特性能够防止未经授权的进程锁定所有可以得到的内存，这必定会导致系统发生停顿（这算作一种 DOS（Denial of Service，拒绝服务）攻击）。

16.2.2 munmap 函数的使用

当你用完一个内存映像文件后，可以调用 `munmap` 解除内存映像并把内存释放返回给操作系统。这个函数的原型如下：

```
#include <unistd.h>
#include <sys/mman.h>
int munmap(void *start, size_t length);
```

参数 `start` 指向要解除映像的内存区的起始位置，而参数 `length` 指出要解除映像的内存区的大小。当一块内存解除映像后，再尝试访问 `start` 会导致一个段错误（产生 `SIGSEGV`）。当一个进程终止运行时，所有的内存映像都被解除。`munmap` 执行成功返回 0，如果执行失败则返回 -1 并且设置 `errno` 变量。

16.2.3 msync 函数的使用

`msync` 函数把被映像的文件写入磁盘。它的原型如下：

```
int msync(const void *start, size_t length, int flags);
```

调用 `msync` 可以用对内存中的映像的更新写入一个被映像的文件。被强行写入到磁盘的内存区从 `start` 指定的地址开始；写入 `length` 个字节的数据。参数 `flags` 可以是下面这些值一个或多个的逻辑“或”。

<code>MS_ASYNC</code>	调度一次写入操作然后返回
<code>MS_SYNC</code>	在 <code>msync</code> 返回前写入数据
<code>MS_INVALIDATE</code>	让映像到同一文件的映像无效，以便用新数据更新它们

16.2.4 mprotect 函数的使用

`mprotect` 函数修改在内存映像上的保护模式。这个函数的原型如下：

```
#include <unistd.h>
#include <sys/mmap.h>
int mprotect(const void *start, size_t len, int prot);
```

mprotect 把自 start 开始的内存区的保护模式修改为 prot 指定的值, prot 可以是表 16.1 列出标志的一个或多个的逻辑“或”。这个函数如果执行成功返回 0。如果执行失败, mprotect 返回-1 并且设置 errno 变量。

16.2.5 锁定内存

如果不深究其本质的工作原理, 内存加锁就意味着防止一块内存区域被交换到磁盘上。在像 Linux 这样的多任务、多用户系统上, 处于不活动状态(一段时间内不被访问)的系统内存(RAM)区可以被暂时地写入(交换到)磁盘, 从而让这部分内存能够用于其他用途。锁定内存就是在内存上设置了一个标志防止它被交换出内存。

有 4 个函数用于对内存加锁和解锁: mlock、mlockall、munlock 和 munlockall。它们的原型如下:

```
#include <unistd.h>
#include <sys/mman.h>
int mlock(const void *start, size_t len);
int munlock(void *start, size_t len);
int mlockall(int flags);
int munlockall(void);
```

start 指出被加锁或解锁的内存区, 而 len 指出加锁或解锁的内存区大小。flags 的值可以是 MCL_CURRENT 和 MCL_FUTURE 之一或两个都有。MCL_CURRENT 在调用返回前请求锁住所有内存页面。MCL_FUTURE 指出锁住所有增加到进程的地址空间的内存页面。正如在讨论 mmap 时所提到的那样, 只有具有超级用户权限的进程才可以对内存区加锁或解锁。

16.2.6 mremap 函数的使用

mremap 函数用于改变一个被映像的文件的大小。这个函数的原型定义如下:

```
#include <unistd.h>
#include <sys/mman.h>
void *mremap(void *old_addr, size_t old_len, size_t new_len,
             unsigned long flags);
```

在偶尔的情况下, 你会需要调整一块内存区的大小, 这也是有这个函数的原因。和前面讨论的 realloc 函数类似, mremap 函数调整自 old_addr 开始的内存区的大小, 把原来的大小 old_len 调整到 new_len。参数 flags 指出如果必要是否在内存中移动内存区。MREMAP_MAYMOVE 允许改动地址; 如果没有指定这个值, 则调整内存区大小的操作执行失败。如果执行成功 mremap 返回内存区调整后的地址, 如果执行失败则返回 NULL。

16.2.7 用内存映像实现 cat 命令

程序清单 16.1 演示了如何使用内存映像文件。虽然这个程序是 cat(1)的一种不完善的

实现，但它还是清楚地展示了内存映像文件的用法。

程序清单 16.1 使用内存映像的 cat 实现

```
/*
 * mmcat.c - Implement the cat command using memory maps
 */
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

void err_quit(char *msg);

int main(int argc, char *argv[])
{
    int fdin;
    char *src;
    struct stat statbuf;
    off_t len;

    /* make sure we were called properly */
    if(argc != 2) {
        fprintf(stderr, "usage: mmcat {file}\n");
        exit(EXIT_FAILURE);
    }

    /* open the input file and stdout */
    if((fdin = open(argv[1], O_RDONLY)) < 0) {
        err_quit("open");
    }

    /* need the size of the input file for mmap call */
    if((fstat(fdin, &statbuf)) < 0) {
        err_quit("fstat");
    }
    len = statbuf.st_size;

    /* map the input file */
    if((src = mmap(0, len, PROT_READ, MAP_SHARED, fdin, 0)) ==
        (void *)-1) {
        err_quit("mmap");
    }

    /* write it out */
    printf("%s", src);
    /* clean up */
    close(fdin);
}
```

```
    munmap(src, len);
    exit(EXIT_SUCCESS);
}

void err_quit(char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}
```

执行 `make mmcat` 编译这个程序。执行这个程序时需给它提供一个文件名，比如 `mmcat mmcat.c`。这个程序中最让人感兴趣的代码片段是对 `fstat` 和 `mmap` 的调用。正如注释中所说明的那样，为了调用 `mmap`，这个程序需要知道输入文件的大小，因此就要调用 `fstat`。一旦文件被映像到了内存中，正如在 `printf("%s", src)` 语句所显示的那样，`mmcat` 能够准确地使用指针 `src`，就好像它是用 `fread` 或 `fgets` 调用得到的。一旦 `mmcat` 用完了映像内存区，程序就用 `munmap` 把它释放回操作系统。工具函数 `err_quit` 减少了程序的代码量。

从实用的角度来看，在本例中使用一个内存映像文件太浪费，因为就其性能和代码量来考虑程序完成的功能太少。但是，在性能成为关键的情况或者当你处理时间密集型操作时，使用内存映像文件就可以带来一定好处。在要求高度安全性的情况下，内存映像也极具价值。因为具有超级用户权限的程序能够把内存映像锁定在内存中，不让它们因 Linux 的内存管理机制而被交换到磁盘上，类似口令文件这样的敏感数据就不大会被扫描程序探测到。当然，在这种场合，内存区应该被设置为 `PROT_NONE`，这样其他进程就不能读这块内存区了。

既然你已经懂得了很多有关内存映像的知识，下一节分析的两个程序能够帮助你调试内存问题。

16.3 发现并修改内存问题

本节介绍的几个工具能帮助你确定代码中的内存管理问题。因为 C 假定编程人员会正确地编写代码，大多数 C 编译器都忽略了诸如使用未初始化内存空间、越上界访问缓冲区、越下界访问缓冲区等错误，而且大多数编译器都不能捕获内存泄漏或悬空指针问题。本节讨论的工具能够弥补这些编译器的缺陷和程序员的失误。

注意： 实际上，大多数编译器都有多种开关和选项让它们能够捕捉到刚才提到的某些错误。例如，GCC 就 `-all` 这个选项（在第 3 章中进行了讨论）。但是，一般说来编译器无法检测到所有的内存问题，这使得本节介绍的工具变得非常有价值。

16.3.1 一个有问题的程序

程序清单 16.2 中的程序带有一些 bug，它们是：

- 有一处内存泄漏
- 越上界访问动态分配的堆内存

- 越下界访问内存缓冲区
- 同一缓冲释放了两次
- 访问已经释放的内存
- 扰乱静态分配的栈和全局内存

程序清单 16.2 一个带有内存 bug 的程序

```
/*
 * badmem.c - Demonstrate usage of memory debugging tools
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char g_buf[5];

int main(void)
{
    char *buf;
    char *leak;
    char l_buf[5];

    /* won't free this */
    leak = malloc(10);

    /* overrun buf a little bit */
    buf = malloc(5);
    strcpy(buf, "abcde");
    printf("LITTLE : %s\n", buf);
    free(buf);

    /* overrun buf a lot */
    buf = malloc(5);
    strcpy(buf, "abcdefgh");
    fprintf("BIG : %s\n", buf);

    /* underrun buf */
    *(buf - 2) = '\0';
    printf("UNDERRUN: %s\n", buf);

    /* free buf twice */
    free(buf);
    free(buf);

    /* access freed memory */
    strcpy(buf, "This will blow up");
    printf("FREED : %s\n", buf);

    /* trash the global variable */
    strcpy(g_buf, "global boom");
    printf("GLOBAL : %s\n", g_buf);
}
```

```
/* trash the local variable */
strcpy(l_buf, "local boom");
printf("LOCAL   : %s\n", l_buf);

exit(EXIT_SUCCESS);
}
```

这些 bug 都不会导致程序不能编译（使用 `make badmem`）通过或者执行，但是内存泄漏和被扰乱的内存通常的症状是在程序的其他位置引起无法预测的行为。在我的系统上，程序的输出为

```
$ ./badmem
LITTLE      : abcde
BIG         : abcdefgh
UNDERRUN    : abcdefgh
Segmentation fault (core dumped)
```

在其他系统上的输出结果可能有所变化。

16.3.2 Electric Fence

本节介绍的内存调试工具是 Electric Fence，它的作者是 Bruce Perens。Electric Fence 不能捕获内存泄漏，但是在检测缓冲越界方面它的表现非常出色。虽然许多 Linux 发布版本都带有这个工具，你也可以从 <ftp://metalab.unc.edu/pub/Linux/devel/lang/c> 得到它。在保存本章源代码目录下子目录 `tools` 里有 2.0.5 版 Electric Fence 的 `tar.gz` 文件。要安装这个工具，可以把文件解压缩，然后在生成的目录中运行 `make install`（当然，用以超级用户身份来执行这个操作）。

提示： 在安装 Electric Fence 之前，要检查你的系统上是否已经安装了这个工具。大多数 Linux 版本都提供了一个 Electric Fence 的拷贝。如果已经安装的 Electric Fence 不是最新的 2.0.5 版，你可能会升级它。

Electric Fence 使用系统的虚拟内存硬件来检测非法的内存访问，并在导致越界的第一条指令上停止运行。它是通过用自己的 `malloc` 函数来替换普通的 `malloc` 函数，然后在被请求的内存位置后面不允许进程访问的地方分配一小段内存来做到这一点的。结果，由于缓冲越界导致内存访问冲突，这会以一个 `SIGSEGV`（段冲突）中止程序的执行。如果你的系统配置允许产生 `core` 文件（执行 `ulimit -c` 取得并设置 `core` 文件允许的大小），随后你就可以使用一个调试器来隔离出发生越界的位置。要使用 Electric Fence，必须把程序和一种特殊的库 `libefence.a` 进行链接：

```
$ gcc -g badmem.c -o badmem -lefence
```

这条编译命令使用 `-g` 选项产生额外的 GDB 兼容的调试符号。当程序执行时，执行中断并且转储内存：

```
$ ./badmem
```

```
Electric Fence 2.0.5 Copyright (C) 1987-1995 Bruce Perens.
```



```
LITTLE : abcde
Segmentation fault (core dumped)
```

接下来，从 GDB 调试器使用 core 文件运行 badmem，使用命令 `gdb badmem`。

```
$ gdb badmem

GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for
details.
This GDB was configured as "i386-COL-linux" ...
(gdb) run
Starting program: /home/kwall/lpu2/16/badmem

Electric Fence 2.0.5 Copyright (c) 1987-1995 Bruce Perens.
LITTLE : abcde

Program received signal SIGSEGV, Segmentation fault.
0x804ab34 in strcpy ()
(gdb) where
#0 0x804ab34 in strcpy()
#1 0x8048215 in main () at badmem.c:27(gdb)
```

上面输出的倒数第二行清楚地表明在 `badmem.c` 中的第 27 行 `main` 函数里发生了一个问题。改正了这个问题之后，你就可以重新编译并运行这个程序，如果执行再次发生中断，重复进行调试/修改/重新编译的过程。当你彻底消除了代码中的所有错误，再次编译时不用和 Electric Fence 链接，调试工作也就结束了。

Electric Fence 捕获了第 27 行的严重的缓冲区越界错误，但是它却忽略的第 21 行的小的缓冲区越界错误。原因何在？导致出现这种特殊现象的原因在于 CPU 分配内存的对齐方式。大多数现代 CPU 都要求内存块和它们本身的字长对齐。例如，Intel x86 的 CPU 要求内存区的起始地址必须能被 4 整除，所以 `malloc` 调用通常返回的是对齐后的内存。Electric Fence 也是如此。请求 5 字节的内存实际上却分配了 8 字节，以便满足内存对齐的要求。所以在第 21 行出现的小的缓冲区越界错误逃过了检测。

幸好 Electric Fence 能够让你通过环境变量 `$EF_ALIGNMENT` 控制它的对齐行为。这个环境变量的默认值是 `sizeof(int)`，但是如果你把它设置为 0，Electric Fence 会检测到更小的缓冲区越界错误。当你把 `$EF_ALIGNMENT` 设置为 0 以后，重新编译并运行这个程序，Electric Fence 就能捕捉到第 21 行出现的小缓冲区越界错误。

```
Program received signal SIGSEGV, Segmentation fault.
0x2ab23db4 in strcpy () from /lib/libc.so.6
(gdb) where
#0 0x2ab23db4 in strcpy () from /lib/libc.so.6
```

```
#1 0x8048932 in main () at badmem.c:21  
(gdb)
```

Electric Fence 还能够识别其他三个控制其行为的环境变量：EF_PROTECT_BELOW=1 用于检测缓冲区下边界越界；EF_PROTECT_FREE=1 用于检测访问已经释放的内存；而 EF_ALLOW_MALLOC_0=1 能让程序分配 0 字节的内存。

16.4 小 结

本章介绍了内存管理工具和技术。它回顾了取得和使用内存区的标准 C 函数，并且还介绍了通过 mmap 系列的系统调用使用内存映像文件。最后，介绍了 Electric Fence，它是一种跟踪内存分配错误的工具。

第 17 章 进程间通信

本章介绍 Linux 用于进程间通信（interprocess communication, IPC）的多种方法。IPC 是描述进程相互通信的方法的一般术语。没有 IPC，进程只能通过文件系统相互交换数据或其他信息，而在进程拥有共同的祖先（比如在执行 fork 之后的父进程/子进程关系）的情况下，只能通过任何被继承的文件描述符相互交换数据或其他信息。特别地，你会学到管道、FIFO、共享内存、信号灯和消息队列。

17.1 管道

本章介绍两种类型的管道：无名管道和有名管道。有名管道通常称作 FIFO（First In, First Out，先入先出）。无名管道没有名字，因为它们从来没有路径名，也从来不会在文件系统中出现。严格地说，它们是和内存中的一个索引节点相关联的两个文件描述符。最后的进程关闭了这些文件描述符中的一个，导致索引节点也就是管道消失。另一方面，有名管道存在于文件系统中。它们称为 FIFO 的原因是由于从中读出数据的顺序和写入数据的顺序相同，所以先进入 FIFO 的数据也先从 FIFO 中出来。图 17.1 展示了无名管道和 FIFO 之间的相似点和不同点。

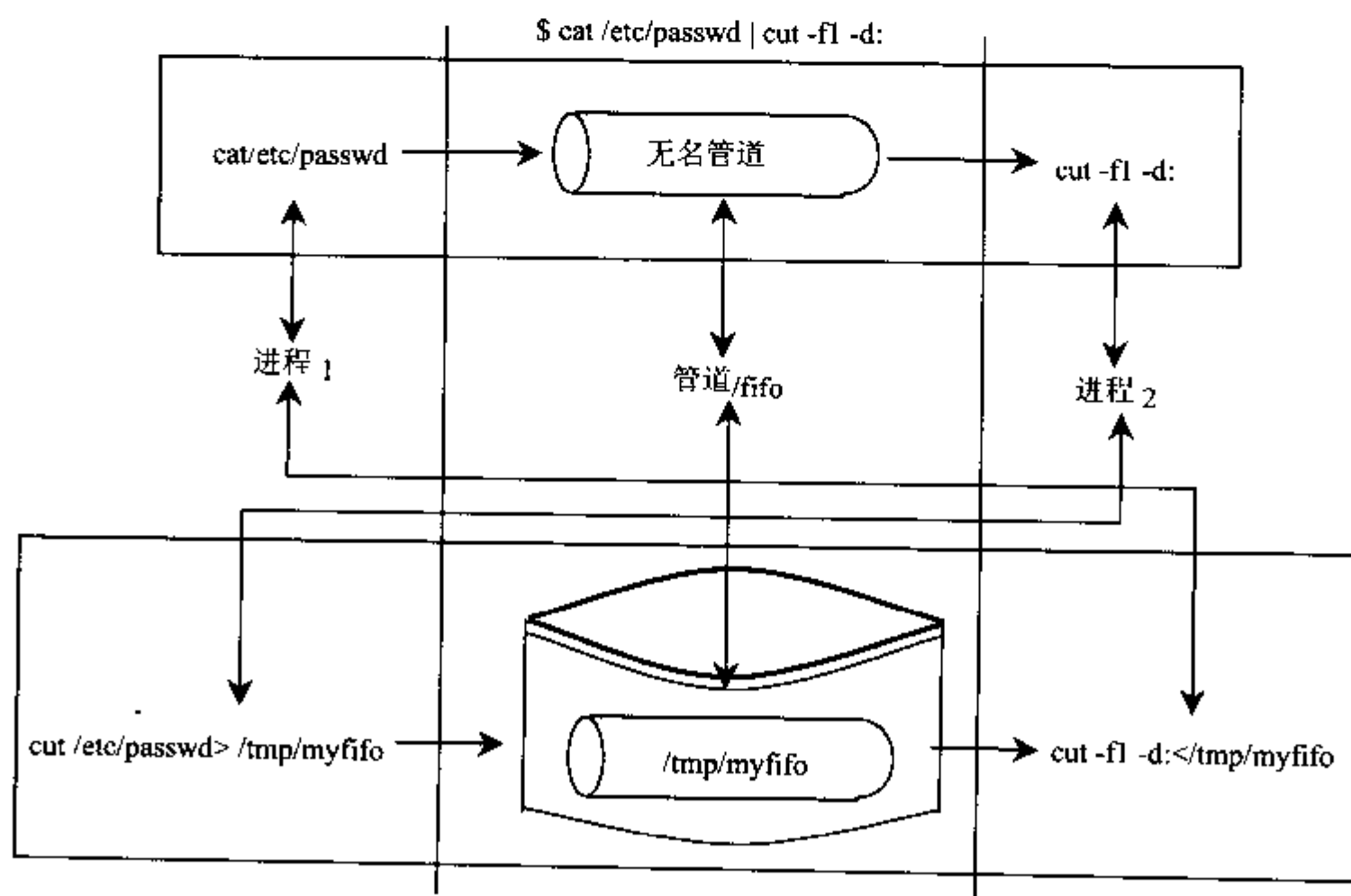


图 17.1 管道和 FIFO 功能类似但又有不同的含义

图 17.1 的上半部分显示了 shell 管道线 `cat /etc/passwd | cut -f1 -d:` 的执行过程。图 17.1 的下半部分显示了如果用有名管道而不是无名管道，上半部分的管道线是怎样执行的。竖线代表数据不是被写入就是被读出一个管道的位置点。双向箭头反映了两种类型管道的输入和输出是怎样相互对应的。这都会在接下来的几节里进行详细介绍，所以可能你在继续阅读本章内容的过程中会反过来查看这幅图。在图的上半部分，`cat` 命令的输出通过内核中的一个无名管道进行传送。无名管道的输出成为了 `cut` 的输入。这是 shell 管道线的典型用法。

特别地，需注意有名管道要求执行 `cat` 和 `cut` 命令的顺序颠倒。后面会解释这样做的原因，必须先执行 `cut` 命令（而且后台运行），这样你就可以在同一终端或从同一控制台发出 `cat` 命令，向有名管道提供输入。

有名管道 `/tmp/myfifo` 和无名管道服务于同样的目的：把它们的输入送给它们的输出。实际上的差别是命令执行的顺序，以及当处理有名管道时要用到 shell 的重定向操作符 “<” 和 “>”。

大多数 Linux 用户都熟悉无名管道，即便他们可能没有意识到这一点。每条像下面这样的命令

```
$ cut -f1 -d: </etc/group | sort
```

都用到了无名管道。在这个例子中，来自 `cut` 命令的输出（它的输入从 `/etc/group` 重定向过来）成为 `sort` 命令的输入。正如你知道的那样，“|” 是管道符号。你没有意识到的一点是可能 shell 会用随后要介绍的 `pipe` 函数实现管道符号 “|”。

注意： 在术语上的一个小技巧：除非需要清楚地进行区别，否则本章里的管道 (pipe) 一词是指无名管道，而 FIFO 一词指有名管道。但是，管道和 FIFO 都是半双工的；也就是说，数据流只有一个方向，就像下水道中的水流只有一个方向一样。管道和 FIFO 也是不能定位的；也就是说，你不能使用像 `lseek` 这样的调用定位文件指针。

无名管道有两个缺点。首先，正如注意中说明的那样，无名管道是半双工的，所以数据只能在一个方向传送。其次，也是更明显的缺点是，管道只能在相关的、有共同祖先的进程之间使用。正如你能回忆起的第 13 章“进程控制”中所介绍的内容，从一个 `fork` 或 `exec` 调用创建的子进程继承了父进程的文件描述符。

17.1.1 打开和关闭管道

自然地，在你从一个管道中读出数据或者向这个管道写入数据之前，这个管道必须存在。创建管道的调用的原型如下：

```
#include <unistd.h>
int pipe(int filedes[2]);
```

如果它成功建立了管道，则会打开两个文件描述符并把它们的值保存在一个整数数组 `filedes` 中。第一个文件描述符 `filedes[0]` 用于读出数据，所以 `pipe` 用 `read` 调用的 `O_RDONLY` 标志打开它。第二个文件描述符 `filedes[1]` 用于写入数据，所以 `pipe` 用 `open` 调用的

O_WRONLY 标志打开它。pipe 执行成功则返回 0，如果出错则返回-1，此时它也会设置全局出错变量 errno。

可能发出的出错条件有 EMFILE，意思是调用进程已经打开了太多的文件描述符，EFAULT，即 filedes 数组无效，或者 ENFILE，意思是内核的文件表满了（这肯定不是个好兆头！）。必须再次强调，这里的文件描述符不能和一个磁盘文件相对应，它只是驻留在内核中的一个索引节点（inode）。图 17.2 显示出了这一点。

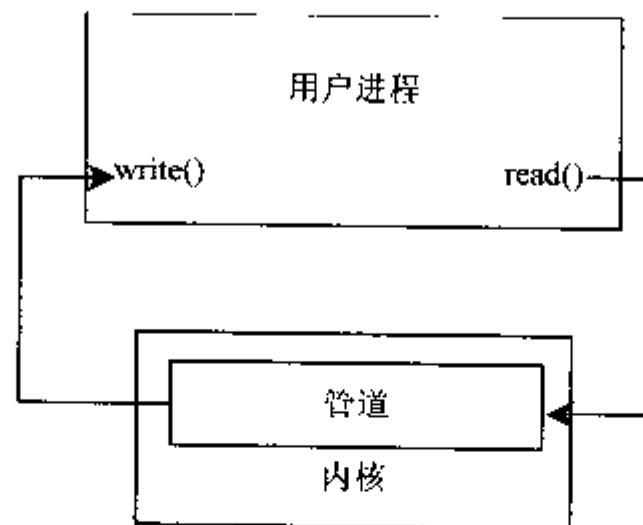


图 17.2 Linux 管道只存在于内核中

要关闭一个管道，可以用系统调用 close 关闭管道所关联的文件描述符。程序清单 17.1 显示了如何打开和关闭一个管道。执行命令 make opipe 使用本书提供的 makefile 文件，编译这个程序。

程序清单 17.1 opipe.c

```

/*
 * opipe.c - Open and close a pipe
 */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int fd[2]; /* Array for file descriptors */
    if((pipe(fd)) < 0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    printf("descriptors are %d, %d\n", fd[0], fd[1]);
    close(fd[0]);
    close(fd[1]);
    exit(EXIT_SUCCESS);
}
  
```

这个程序的输出（见后）显示 pipe 调用成功（在你的系统上，文件描述符的值可能有所不同）。这个程序调用 pipe 函数，给它传递了一个文件描述符数组 fd。如果 pipe 调用成功，则程序打印输出文件描述符的整数值，再把它们两个都关闭，然后退出。

运行这个程序产生的输出应该和下面的结果类似：

```
$. /opaque
descriptors are 3, 4
$
```

17.1.2 读写管道

要从管道中读出数据和向管道写入数据，只需简单地使用 `read` 和 `write` 调用即可。请记住，`read` 从 `filedes[0]` 中读出数据，而 `write` 向 `filedes[1]` 写入数据。

也就是说，几乎不会在一个进程中打开一个管道仅供进程自己使用。管道是用来交换数据的。因为一个进程已经能够访问它要通过管道共享的数据，和它自己共享数据是没有意义的。按照常规，一个进程调用 `pipe` 然后再调用 `fork` 产生一个子进程。因为子进程从父进程继承了任何打开的文件描述符，就建立起了一个 IPC 通道。下一步取决于进程是读数据还是写数据。一般的规则是读数据的进程关闭管道的写入端，而写数据的进程关闭管道的读出端。下面两条叙述更明确地说明了上述过程：

- 如果父进程正在向子进程发送数据，则父进程关闭 `filedes[0]` 并向 `filedes[1]` 写入数据，而子进程关闭 `filedes[1]` 并从 `filedes[0]` 读出数据。
- 如果子进程把数据返回给父进程，则子进程关闭 `filedes[0]` 并向 `filedes[1]` 写入数据，而父进程关闭 `filedes[1]` 并从 `filedes[0]` 读出数据。

图 17.3 应能帮助你想像正确的步骤并记住这个规则。

警告： 试图对一个管道的两端进行读写操作是一个严重的编程错误。如果两个进程需要全双工的管道功能，父进程必须在 `fork` 之前打开两个管道。

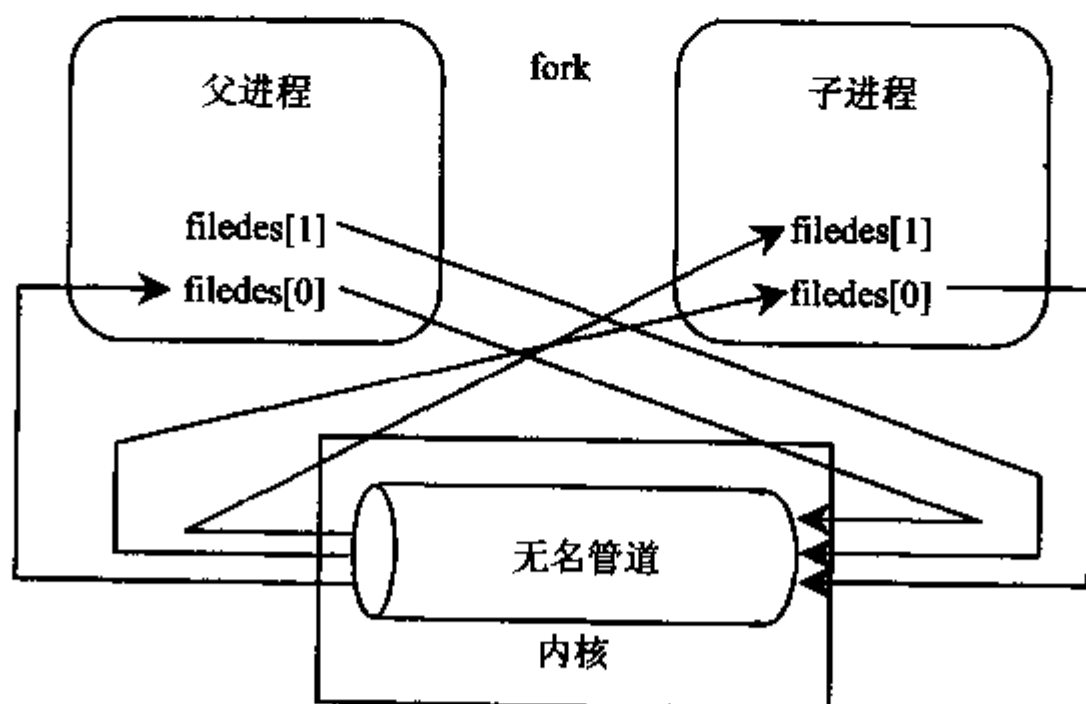


图 17.3 在 `fork` 之后读写管道

图 17.3 的上部显示出 `fork` 之后文件描述符的部署：父进程和子进程中都有这两个打开的文件描述符。图 17.3 假定父进程写数据而子进程读数据。图中的下半部分显示了在父进程关闭了它的读描述符和子进程关闭了它的写描述符后的文件描述符状态。

注意： 在关闭管道的写入端后，从管道读数据会返回 0 以指示文件末尾。但是，如果关闭读取端，任何写入管道的尝试都会给写入方产生 SIGPIPE 信号，而 write 调用自己返回 -1 并设置全局变量 errno 的值为 EPIPE。如果写入方不能捕获或者干脆忽略 SIGPIPE，则写入进程会中断。

程序清单 17.2 中的程序 piperw.c 显示了在相关进程间打开一个管道的正确步骤。执行 make piperw 编译这个程序。

程序清单 17.2 piperw.c

```
/*
 * piperw.c - The correct way to open a pipe and fork
 *            a child process.
 */
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>

#define BUFSZ PIPE_BUF

void err_quit(char *msg);

int main(int argc, char *argv[])
{
    int fd[2]; /* File descriptor array for the pipe */
    int fdin; /* Descriptor for input file */
    char buf[BUFSZ];
    int pid, len;

    /* Create the pipe */
    if((pipe(fd)) < 0)
        err_quit("pipe");

    /* Fork and close the appropriate descriptors */
    if((pid = fork()) < 0)
        err_quit("fork");
    if (pid == 0) {
        /* Child is reader, close the write descriptor */
        close(fd[1]);
        while((len = read(fd[0], buf, BUFSZ)) > 0)
            write(STDOUT_FILENO, buf, len);
        close(fd[0]);
    } else {
        /* Parent is writer, close the read descriptor */
        close(fd[0]);
        if((fdin = open(argv[1], O_RDONLY)) < 0) {
            perror("open");
        }
    }
}
```

```

        /* Send something since we couldn't open the input */
        write(fd[1], "123\n", 4);
    } else {
        while((len = read(fdin, buf, BUFSZ)) > 0)
            write(fd[1], buf, len);
        close(fdin);
    }
    /* Close the write descriptor */
    close(fd[1]);
}
/* Reap the exit status */
waitpid(pid, NULL, 0);
exit(EXIT_SUCCESS);
}

void err_quit(char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}

```

`piperw` 需要一个输入文件的名称否则它向读取进程发送 `123\n`。使用 `opipe.c` 作为输入的一次示范运行产生的输出如下：

```

$ ./piperw opipe.c
/*
 * opipe.c - Open and close a pipe
 */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int fd[2]; /* Array for file descriptors */

    if((pipe(fd)) < 0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    printf("descriptors are %d, %d/n", fd[0], fd[1]);
    close (fd[0]);
    close(fd[1]);
    exit(EXIT_SUCCESS);
}
$

```

正如你能从输出中看到的那样，除了 `piperw` 使用管道而不是简单地把输入显示在标准输出 `stdout` 上之外，它的行为更像是 `cat` 命令。在 `fork` 之后，子进程关闭了它所继承的供

写入的文件描述符，因为它只从管道读取数据。类似地，父进程写入数据，所以它关闭了它的读取文件描述符。

如果父进程不能打开它的输入文件 (`argv[1]`)，在它关闭前，它还通过管道发送字符串 `123\n`，而不是马上终止程序。一旦父进程通过管道向子进程传完数据，它就关闭它的写描述符。当子进程从管道读到 0 字节时，它就关闭读描述符。最后，即使还不清楚是父进程还是子进程先退出，但是父进程调用 `waitpid` 取得子进程的退出状态以避免产生一个僵进程或者孤儿进程。

注意： 如果多个进程正在向同一个管道写入数据，每个进程的写入的数据必须少于 `PIPE_BUF` 个字节，它是在 `<limits.h>` 中定义的宏，以保证是原子写操作 (atomic write)；也就是说，一个进程写入的数据不能和另一个进程写入的数据相混合。用一条原则说明这个问题，为了确保执行原子写操作，要限制每次 `write` 调用写入的数据量少于 `PIPE_BUF` 个字节。

17.1.3 更简单的方法

`piperw` 程序为了实现 `cat` 一个文件这一功能做了许多工作：创建一个管道，`fork`，在父进程和子进程中关闭不需要的文件描述符，打开一个输入文件，从管道读出和向管道写入数据，关闭打开的文件和文件描述符，接着还要取得了子进程的退出状态。因为这一操作顺序很常用，所以 ANSI/ISO C 已经把它们编入了标准库函数 `popen` 和 `pclose` 中，它们的原型如下：

```
#include <stdio.h>
FILE *popen(const char *command, const char *mode);
int pclose(FILE *stream);
```

`popen` 创建一个管道然后 `fork` 出一个子进程，接着执行一个 `exec` 调用，调用 `/bin/sh -c` 执行保存在 `command` 中的命令字符串。参数 `mode` 必须是 `r` 或 `w`，在这里它们和在标准 I/O 库中的语义相同。也就是说，如果 `mode` 是 `r`，`popen` 返回的 `FILE` 流指针被打开供读数据所用，这意味着这个流被附加到 `command` 的标准输出；从这个流读出数据和读 `command` 的标准输出是一样的。同样，如果 `mode` 是 `w`，这个流被附加到 `command` 的标准输入，所以向这个流写入数据和向 `command` 的标准输入写入数据是一样的。如果 `popen` 调用失败，则返回 `NULL`。导致失败的出错条件被设置在出错变量 `errno` 中。

为了关闭这个流，可以使用 `pclose`。`pclose` 关闭 I/O 流，等待 `command` 来完成，并向调用进程返回它的退出状态。如果 `pclose` 调用失败，则返回 `-1`。

程序清单 17.3 中的程序 `popen.c`（使用命令 `make popen` 编译）用 `popen` 和 `pclose` 重写了 `piperw` 程序。

程序清单 17.3 `popen.c`

```
/*
 * popen.c - Using popen() to open a pipe
 */
#include <unistd.h>
```

```

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>

#define BUFSZ PIPE_BUF

void err_quit(char *msg);

int main(void)
{
    FILE *fp; /* FILE stream for popen */
    char *cmdstring = "cat popen.c";
    char buf[BUFSZ]; /* Buffer for "input" */

    /* Create the pipe */
    if((fp = popen(cmdstring, "r")) == NULL)
        err_quit("popen");

    /* Read cmdstring's output */
    while((fgets(buf, BUFSZ, fp)) != NULL)
        printf("%s", buf);

    /* Close and frap the exit status */
    pclose(fp);
    exit(EXIT_SUCCESS);
}

void err_quit(char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}

```

正如你能从程序清单中看到的那样，`popen` 和 `pclose` 使用管道来工作用到的代码更少。这样做的代价是放弃了一定程度的控制能力。例如，限定编程人员只能用 C 的流库而不能使用低级的 `read` 和 `write` I/O 调用。另外，`popen` 迫使你的程序在这里执行一次 `exec` 调用，这可能是你不想或不需要做的。最后，函数调用 `pclose` 通常要取得子进程的退出状态，这也可能不符合你的编程要求。虽然损失了灵活性，但是 `popen` 节省了自身 10~15 行代码，这大大简化了程序 `pipew` 中处理读写的代码。下面的例子显示，`pipew` 的输出没有改变。而参数 `mode` 的语义违反了人们的直觉，所以要记住 `r` 的意思是你从 `stdout` 读取数据，而 `w` 的意思是你向 `stdin` 写入数据。

```

$ ./popen
/*
 * popen.c - Using popen() to open a pipe
 */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

```

```
#include <fcntl.h>
#include <limits.h>

#define BUFSZ PIPE_BUF

void err_quit(char *msg);

int main(void)
{
    FILE *fp; /* FILE stream for popen */
    char *cmdstring = "cat popen.c";
    char buf[BUFSZ]; /* Buffer for "input" */

    /* Create the pipe */
    if((fp = popen(cmdstring, "r")) == NULL)
        err_quit("popen");

    /* Read cmdstring's output */
    while((fgets(buf, BUFSZ, fp)) != NULL)
        printf("%s", buf);

    /* Close and reap the exit status */
    pclose(fp);
    exit(EXIT_SUCCESS);
}

void err_quit(char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}
$
```

17.2 FIFO

正如前面所介绍的那样，FIFO 也称为有名管道，因为它们是持久稳定的；也就是说，它们存在于文件系统中。FIFO 比无名管道作用更大，因为它们能让无关联的进程交换数据，而且更因为它们能永久地驻留在文件系统中。

17.2.1 理解 FIFO

一个使用 shell 命令的简单例子能够帮助你理解 FIFO。mkfifo(1)命令创建 FIFO：

```
mkfifo [option] name [...]
```

mkfifo 创建一个名为 name 的 FIFO。option 通常为 -m mode，这里的 mode 指出新创建的 FIFO 的八进制模式，它会受到调用进程的 umask 的修正。一旦你创建好了 FIFO，就可以像一个普通的 shell 管道线那样使用它。

下面的例子通过向一个创建时名为 fifo1 的 FIFO 发送 popen 的输出，然后 FIFO 又通

过 `cut` 命令发送自己的输出。

首先，使用下面的命令创建 FIFO：

```
$ mkfifo -m 600 fifol
```

接下来，在建立 `popen` 之后，执行下面两条命令：

```
$ cat < fifol | cut -c1 -5 &
$ ./popen > fifol
```

这些 shell 命令的输出如下：

```
/*
 * po
 */
#include
#include
#include
#include
#include
#include

#define

void

int m
{
    F
    c
    c

    /
    I

    /
    w

    /
    p
    e
}

void
{
    p
    exit
}
[1]+  Done                  cat <fifol | cut -c1 -5
$
```

在后台运行的 `cat` 命令从 FIFO `fifol` 读入自己的输入。`cat` 的输出是 `cut` 命令的输入，`cut` 命令把每一行的输入除了前 5 个字符之外全部裁剪掉了。最后，`cat` 的输入是程序 `popen` 的输出。

这些 shell 命令的实际输出是被裁剪的代码，样子看上去挺奇怪。如果 `fifo1` 是个普通文件，那么最后的结果只会是产生一个填满了 `popen` 输出的普通文件。

17.2.2 创建 FIFO

创建 FIFO 的函数调用和 shell 界面创建 FIFO 的命令名称相同，都是 `mkfifo`。它的语法和系统调用 `open` 的语法很相似：

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

`mkfifo` 用 `mode`（以八进制）指定的权限位创建一个名为 `pathname` 的 FIFO。通常，`mode` 中的值会被进程的 `umask` 修改。

注意：要事先判断一个文件或目录在被 `umask` 修改了之后权限模式是什么，只需把你所使用的模式和 `umask` 的反码进行逻辑“与”操作就可以了。以代码的形式说明，就类似于下面：

```
mode_t mode=0666;
mode & ~umask;
```

于是对于 `umask` 为 022 的情况来说，`mode & ~umask` 返回值为 0644。

如果执行成功，`mkfifo` 返回 0。否则，它设置出错变量 `errno` 的值并且向调用函数返回 -1。变量 `errno` 潜在的出错值包括 `EACCESS`、`EEXIST`、`ENAMETOOLONG`、`ENOENT`、`ENOSPC`、`ENOTDIR` 和 `EROFS`。

程序清单 17.4 在当前目录下创建一个 FIFO。

程序清单 17.4 newfifo.c

```
/*
 * newfifo.c - Create a FIFO
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    mode_t mode = 0666;

    if(argc != 2) {
        puts("USAGE: newfifo {name}");
        exit(EXIT_FAILURE);
    }

    if((mkfifo(argv[1], mode)) < 0) {
```

```
        perror("mkfifo");  
        exit(EXIT_FAILURE);  
    }  
    exit(EXIT_SUCCESS);  
}
```

执行 `make newfifo` 编译这个程序。如果你紧跟着前一个例子，那么要确保在执行 `newfifo` 之前删除 `fifo1`。这个程序几次运行产生的结果如下：

```
$/newfifo  
USAGE: newfifo <name>  
$ ./newfifo fifo1  
$ ./newfifo fifo1  
mkfifo: File exists
```

第一次运行 `newfifo` 发出抱怨是因为没有正确调用它；它要求传递一个 FIFO 的名字作为它惟一的参数。第二次运行提供了一个文件名，所以 `newfifo` 默默地创建了这个文件。因为文件已经存在，第三次运行中的 `mkfifo` 调用执行失败，`errno` 被设置为 `EEXIST`。`EEXIST` 的情况和 `perror` 打印的字符串：`mkfifo: File exists` 相对应。

17.2.3 打开和关闭 FIFO

打开、关闭、删除、读取和写入 FIFO 分别使用系统调用 `open`、`close`、`unlink`、`read` 和 `write` 来完成，它们体现出了你已经见过的——Linux 的“一切都是文件”这一抽象概念的优点。因为打开和关闭 FIFO 等同于打开和关闭管道，你可能要复习一下在 17.1.1 节中给出的打开和关闭管道的程序。

但是在你读写 FIFO 的时候必须留意几点。第一点，FIFO 的两端都必须在使用之前打开。第二点更重要，要留意以 `O_NONBLOCK` 标志打开的 FIFO 的行为。记得 `O_WRONLY` 或 `O_RDONLY` 标志都可以和 `O_NONBLOCK` 标志进行逻辑“或”。如果 FIFO 是以 `O_NONBLOCK` 和 `O_RDONLY` 标志打开的，那么调用会立即返回，但如果它是以 `O_NONBLOCK` 和 `O_WRONLY` 标志打开的，在 FIFO 还没有为读出数据打开时，调用 `open` 会返回一个错误，并且把变量 `errno` 设置为 `ENXIO`。

另一方面，如果在 `open` 调用的标志中没有指定 `O_NONBLOCK`，`O_RDONLY` 将阻塞 `open` 调用（不返回）直到另一个进程为写入数据打开 FIFO 为止。类似地，`O_WRONLY` 也导致阻塞直至为读出数据打开 FIFO 为止。

和使用管道的情况类似，向一个没有为读出数据打开的 FIFO 写入数据会导致向写入进程发送 `SIGPIPE` 信号并且设置变量 `errno` 为 `EPIPE`。在最后一个写数据的进程关闭 FIFO 以后，读数据的进程能在它下一次的读操作中检测到 EOF 标志。正如在和管道有关的章节中提到的那样，为了保证多个进程向一个 FIFO 写数据时执行的是原子写操作，每个进程的写操作都必须写入少于 `PIPE_BUF` 个字节的数据。

17.2.4 读写 FIFO

受上一节结束时讨论的影响，读写 FIFO 和读写管道以及读写普通文件非常相似。

下面的例子有点复杂。程序清单 17.5 中的 `rdfifo` 创建并打开了一个 FIFO 供读取数据使用，将 FIFO 的输出显示在标准输出设备 `stdout` 上。下一个程序，即程序清单 17.6 中的 `wrfifo` 打开 FIFO 供写数据使用。特别有趣的是，同时在不同的窗口运行几个写进程的实例并且观察它们在运行读进程的窗口中的输出。

程序清单 17.5 `rdfifo.c`

```
/*
 * rdfifo.c - Create a FIFO and read from it
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>

int main(void)
{
    int fd; /* Descriptor for FIFO */
    int len; /* Bytes read from FIFO */
    char buf[PIPE_BUF];
    mode_t mode = 0666;

    if((mkfifo("fifo1", mode)) < 0) {
        perror("mkfifo");
        exit(EXIT_FAILURE);
    }

    /* Open the FIFO read-only */
    if((fd = open("fifo1", O_RDONLY)) < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    /* Read and display the FIFO's output until EOF */
    while((len = read(fd, buf, PIPE_BUF - 1)) > 0)
        printf("rdfifo read: %s", buf);
    close(fd);

    exit(EXIT_SUCCESS);
}
```

程序清单 17.6 `wrfifo.c`

```
/*
 * wrfifo.c - Write to a "well-known" FIFO
 */
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>
#include <time.h>

int main(void)
{
    int fd;          /* Descriptor for FIFO */
    int len;          /* Bytes written to FIFO */
    char buf[PIPE_BUF]; /* Ensure atomic writes */
    time_t tp;        /* For time call */

    /* Identify myself */
    printf("I am %d\n", getpid());

    /* Open the FIFO write-only */
    if((fd = open("fifo1", O_WRONLY)) < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    /* Generate some data to write */
    while(1) {
        /* Get the current time */
        time(&tp);

        /* Create the string to write */
        len = sprintf(buf, "wrfifo %d sends %s", getpid(), ctime(&tp));

        /*
         * Use (len + 1) because sprintf does not count
         * the terminating null
         */
        if((write(fd, buf, len + 1)) < 0) {
            perror("write");
            close(fd);
            exit(EXIT_FAILURE);
        }
        sleep(3);
    }

    close(fd);
    exit(EXIT_SUCCESS);
}
```

你可以在包含本章源代码的目录下执行 `make rdfifo wrfifo` 来编译这两个程序。这两个程序的输出如图 17.4 所示。读进程 `rdfifo` 运行在一个大的 `xterm` 窗口中。为了重新创建这个例子，如果存在 `fifo1` 就先删除它。接下来打开四个 `xterm` 窗口。在一个 `xterm` 窗口里启

动 `rdfifo`。然后在另外三个窗口里执行 `wrfifo`。这三个 `xterm` 小窗口每个都运行着一个写程序 `wrfifo` 的实例。每个写进程的 PID 都显示在它的窗口中。每隔 2 秒，写进程把包含它们 PID 和当前时间的消息送进 FIFO，即 `fifo1` 中。正如你所看到的那样，读进程显示接收到的消息，并在消息前面加上 `rdfifo read` 字样，以区分 `rdfifo` 的输入和从 FIFO 读进的输入。

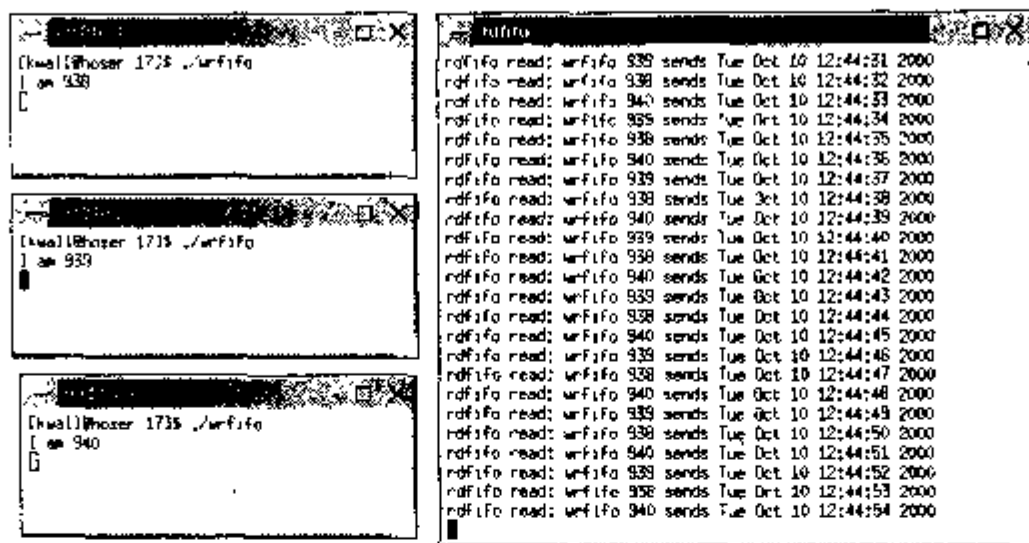


图 17.4 写到同一 FIFO 的多进程

值得注意的是，这些程序构成了一种原始的客户机/服务器应用。服务器是 `rdfifo`；它处理经 FIFO 发送给它的消息。客户机是几个 `wrfifo` 进程，它们所扮演的角色是向服务器发送消息。

更高级的客户机/服务器应用会对接收到的数据执行某种处理并且向客户机发回某种通知或数据。深入介绍客户机/服务器应用的知识超出了本书的范围。然而，你确实编写了一个客户机-服务器应用程序。

17.3 System V IPC 概述

在新的应用中很少会用到 System V IPC，因为它已经被 POSIX IPC 取代了。然而本书仍然要介绍它是因为你很有可能会在老程序中遇到它，编写这些老程序时 POSIX IPC 标准还没有出现呢。所有三种 System V IPC 都有着相同的基本接口和相同的一般设计。本节介绍 System V IPC 的基本概念，并且考察信号灯、消息队列以及共享内存相同的特性和编程规范。

17.3.1 System V IPC 的主要概念

像管道一样，IPC（信号灯、消息队列和共享内存）存在于内核而不是像 FIFO 一样存在于文件系统中。IPC 的几种结构有时合起来叫做 IPC 对象，这样可以避免笨拙地说或者写“信号灯、消息队列和共享内存区”这一大段话。类似地，“IPC 对象”一词也常指三种结构类型中的一种，而不必特指到底是哪一种。图 17.5 显示出无关进程间怎样通过一个 IPC 对象相互进行通信。

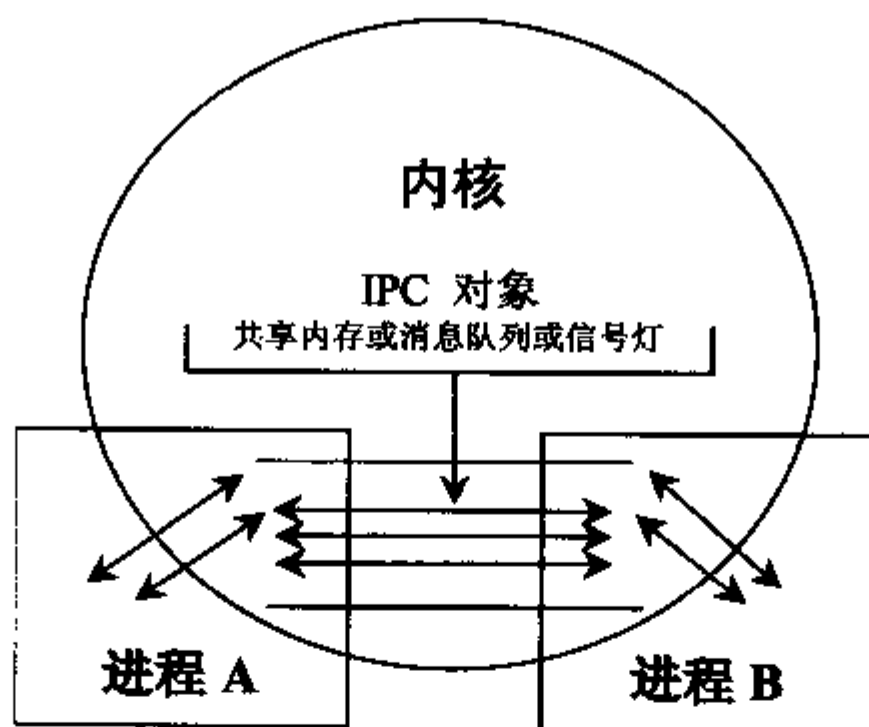


图 17.5 IPC 对象让无关进程交换数据成为可能

正如你在图 17.5 中看到的那样，IPC 对象保存在内核（实际上是内核内存）中，它让其他无关进程（没有相同父进程的进程）通过一种 IPC 机制（共享内存、信号灯或消息队列）相互进行通信。数据在使用 IPC 机制的进程间自由流动。

每个对象都通过它的标识符来引用和访问，标识符是一个正整数，它唯一地标识出对象本身和它的类型。每个标识符的类型都是惟一的，但是同一标识符的值可以用于一个消息队列、一个信号灯和一个共享内存区。标识符成为该结构上所有其他操作的句柄。IPC 结构标识符不像文件描述符那样使用较小的正整数。实际上，随着结构的创建和删除，标识符的值（正式的名称叫做槽使用顺序号）会不断增加直至达到一个最大值为止，然后再转回到 0 并重新开始。在 Linux 系统中，标识符声明为整数，所以它的值最大可能为 65 535。

每个 IPC 结构都由 `get` 函数创建：`semget` 创建信号灯、`msgget` 创建消息队列而 `shmget` 创建共享内存的结构。每次用其中一种 `get` 函数创建对象时，调用函数必须指定一种关键字（`key`），它的类型（在 `<sys/types.h>` 中声明）是内核用来产生标识符的 `key_t` 类型。Linux 2.2.x 版的内核定义 `key_t` 类型是一个整数。

在创建了一个 IPC 结构之后，使用同一关键字的 `get` 函数的后续调用不会创建新结构，但返回和现有结构相关的标识符。这可以让两个或两个以上的进程用同一关键字调用 `get` 函数以建立一条 IPC 通道。

接下来的问题是怎样确保所有要使用同一 IPC 结构的进程都使用相同的关键字。一种方法为，实际创建结构的进程给 `get` 函数传递 `IPC_PRIVATE` 关键字，这能够保证创建一个新结构。然后创建 IPC 结构的进程把返回的标识符保存在其他进程能够访问的文件系统中。在父进程 `fork` 和 `exec` 一个新子进程的场合里，父进程可以把返回的标识符作为一个参数传递给创建子进程的函数 `exec`。

另一种传递关键字的方法是把它保存在公共的头文件中，这样一来所有包含了这个头文件的程序都能够访问到相同的关键字。这种方法引出的一个问题，没有进程知道它是正在创建一个新结构呢还是只访问已经由其他进程创建好的结构。这种方法带来的另一个问题是关键字可能已经被另一个无关的程序使用了。结果使用这个关键字的进程必须包含处理这种可能性的代码。

第三种方法使用了 `flock` 函数，这个函数接受一个路径名和一个称为项目标识符的单个字符作为参数，返回一个关键字可以传递给合适的 `get` 函数。有程序员负责保证让所有的进程事先知道路径名和项目标识符。你可以使用前面提到过的方法之一：在公共的头文件中包含路径名和项目标识符，或者把它们保存在预定义的配置文件中，来做到这一点。

不幸的是，`flock` 有个严重的缺陷：它不能保证产生惟一的關鍵字，这一来出现了和前面讨论的第二种方法一样的问题。因为使用 `flock` 具有潜在的问题，所以本章不讨论它。在下列情况下，`flock` 生成惟一的關鍵字：

- 当两个不同的符号链接链接到同一个文件上。
- 当路径名的索引节点的前 16 个比特位具有相同的值。
- 当系统带有两个具有相同次设备号的硬盘时，在系统有多个磁盘控制器的情况下才会出现。主设备号不能相同，但次设备号可以相同。

考虑到 Linux 的 `flock` 实现有弱点，所以强烈建议读者不使用它并且忽略它。

除了指定一个关键字之外，`get` 函数还要接受一个控制 `get` 行为的 `flag` 参数。如果指定的关键字还没有用于所期望类型的结构，而且在 `flag` 中设置了 `IPC_CREAT` 位，则创建一个新结构。

每个 IPC 结构都有一个模式 (`mode`)，它是类似于文件模式 (和传递给 `open` 调用的情况类似) 的权限集合，区别在于 IPC 结构没有执行权限的概念。当创建一个 IPC 结构的时候，你必须使用为系统调用 `open` 和 `creat` 规定的八进制记法，对特定的权限按位进行“或”操作，并把结果放入参数 `flag` 并传递给 `get` 函数，否则的话，你可能不能访问创建好的结构。以后你将遇到针对它的例子。正如你所期望的那样，System V IPC 包含了一个调用，用来改变 IPC 结构的访问权限和所有权。

17.3.2 System V IPC 的问题

System V IPC 有几个缺点。第一，和它提供的好处相比，其编程接口过于复杂。第二，IPC 结构比其他资源，比如系统能够支持的文件数量或者系统允许的活动进程数目等受到的限制大得多。第三，尽管是一种受限资源，但 IPC 结构却没有保留一个引用计数，它是一个记录使用结构的进程数目的计数器。因此，System V IPC 没有回收被丢弃的 IPC 结构的自动机制。

例如，如果一个进程创建了一个结构，在其中放入数据，然后没有正常地删除结构及其相关数据就终止执行了，那么这个结构会保留在原地直至发生下面一种情况：

- 系统重启。
- 使用 `ipcsrm` 命令专门删除。
- 具有适当访问权限的另一个进程既可以读取数据也可以删除它，也可以两种操作都执行。

最后，正如在前面说明的那样，IPC 结构只存在于内核中；文件系统不知道它们。因此，对它们的 I/O 操作需要学习另一种编程接口。没有文件描述符，你 cannot 通过系统调用 `select` 使用多路复用的 I/O。如果进程需要等待 IPC 结构上的 I/O 操作，它必须使用某种忙

-等待循环。一个忙-等待循环——持续检查某些变化条件的循环——几乎在任何时刻都是一种糟糕的编程习惯，因为它不必要地消耗了 CPU 周期。在 Linux 下，忙-等待循环特别有害，有几种办法，比如阻塞 I/O、使用系统调用 `select` 和信号来实现非忙-等待的循环。

17.3.3 Linux 和 System V IPC

System V IPC 很有名也很常用，但是它的 Linux 实现却很不完善。Linux 版本也提前提供了 POSIX IPC，但即使 2.2.x 版的内核能够支持 POSIX IPC，也几乎没有 Linux 程序真正实现它。POSIX IPC 和本章以及前面章节讨论的早先的 System V IPC 有类似接口，但它既消除了 System V 已有的一些问题，也简化了接口。问题在于，虽然 System V IPC 是标准，但它在 Linux 下实现得很糟糕。

这种局面的后果就是，坚持 POSIX 兼容（绝大多数情况下很成功）的 Linux 既实现了一种 POSIX IPC 的早期版本，也实现了 System V 版本。困难在于 System V 的版本已经完全建好了而且更常用，但 POSIX 的版本更好也易于使用，并且对三种类型的 IPC 对象有更为统一的编程接口。选择什么样的结果呢？我选择打破自己的规则，介绍你可能会在现有的和新的程序中遇到的东西，而不是讨论“正确的东西”；也就是说，POSIX IPC。

当处理信号灯时出现了另外一个问题。System V 的信号灯是在“黑暗时代”创造的，在一个进程（和多个进程）中的多道执行线程需要几乎同时地访问相同的系统资源的时候就会引发许多问题。有经验的程序员会在使用多线程之前先寻求其他解决方案，因为多线程难以正确地实现。

本章对信号灯的讨论被简化了，因为 System V 版本是由多线程创建的。虽然如此，正如你将在本章看到的那样，信号灯在标准的、单线程的程序中相当有用。POSIX 信号灯接口更简单可现在在 Linux 程序中应用不广。

17.4 共享内存

共享内存是你将学习的真正 IPC 三种类型的第一种。这三种类型合起来称为 System V IPC，因为它们源自于最初由 AT&T 发布的 System V UNIX。源自于 BSD 的 UNIX 实现以及其他类似 UNIX 的操作系统（包括 Linux）也支持它们。

共享内存是由内核出于在多个进程间交换信息的目的而留出的一块内存区（段）。如果段的权限设置恰当，每个要访问该段内存的进程都可以把它映像到自己私有的地址空间中。如果一个进程更新了段中的数据，那么其他进程立即会看到更新。由一个进程创建的段也可以由另一个进程读写。共享内存这一名称表达出是由多个进程分享对段及其保存的数据的访问权这一含义。

每个进程都把它自己对共享内存的映像放入自己的地址空间。实际上，从概念上看，共享内存很像内存映像文件。图 17.6 显示了共享内存。

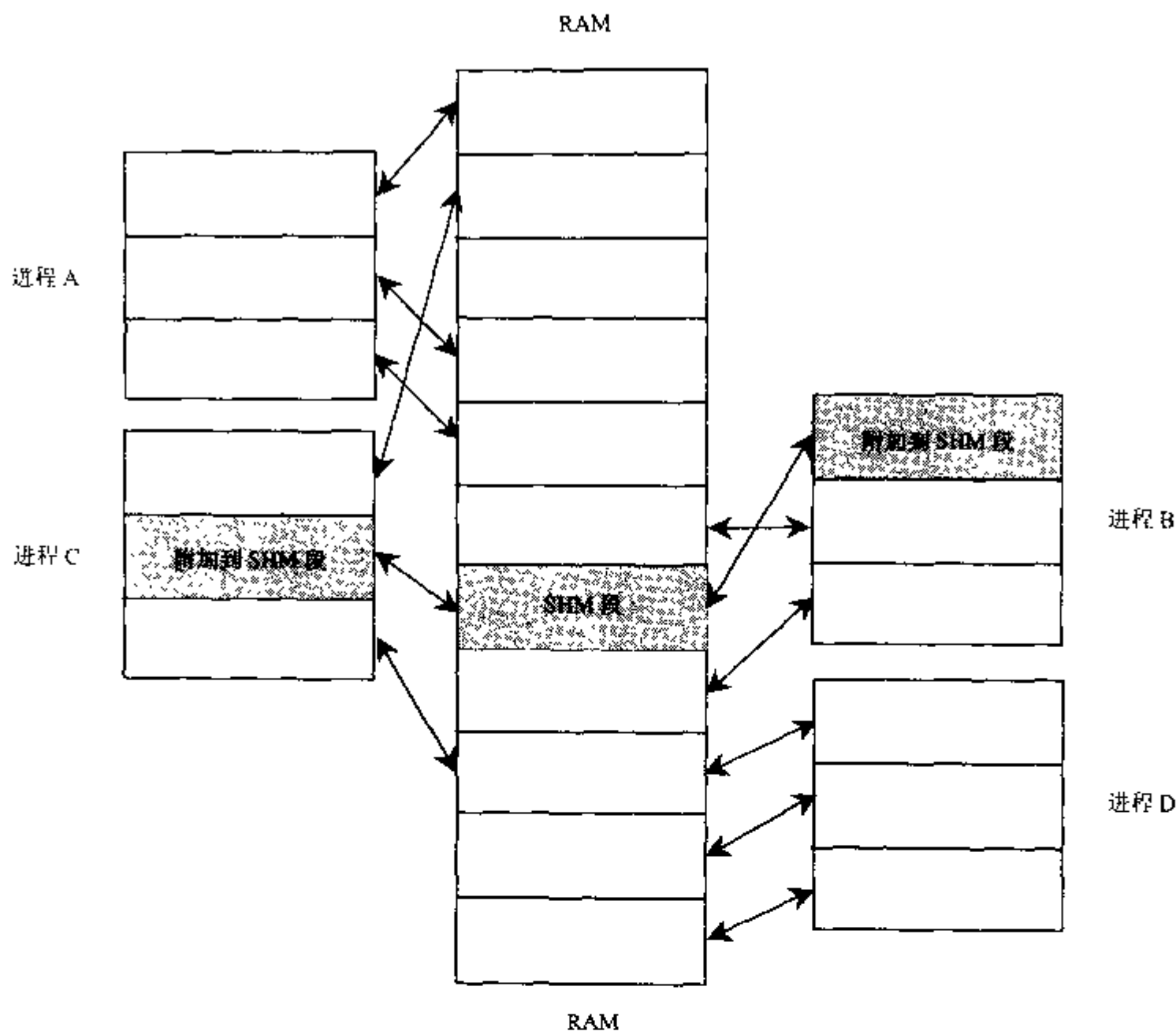


图 17.6 进程把共享内存区映像到它们自己的地址空间

图 17.6 有点过于简化地描绘了共享内存的概念，因为共享内存区是由物理内存中的数据和已经被交换到磁盘上的内存页构成的。附加到共享内存区的进程地址空间同样也应该注意这一点。

不过图中还是显示了在主存中创建的一个共享内存（SHM）段（图中的阴影块）。进程 B 和 C 的阴影块表明这两个程序已经把这段内存映像到自己的地址空间中了。图中还显示了四个进程的每一个都拥有自己的映射到物理 RAM 的地址空间。但是一个进程的地址空间不能被其他进程使用。

自然地，因为数据传送严格地只发生在内存中间（忽略一个或多个内存页可以被交换到磁盘的可能性），共享内存是两个进程间一种快速的通信方式。它具有许多和内存映像文件一样的优点。

17.4.1 创建共享内存区

创建一块共享内存区的调用是 `shmget`。`shmget` 的原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
int shmget(key_t key, int size, int flags);
```

flags 可以是一个或多个 IPC_CREAT、IPC_EXCL 和一组权限位（模式）按位“或”的结果。权限位以八进制表示。IPC_EXCL 确保如果段已经存在，调用执行失败而不是返回一个已经存在的段的标识符。

IPC_CREAT 指出如果没有和 key 关联的段就应该创建一个新段。key 既可以是 IPC_PRIVATE 也可以是 flock 函数返回的一个关键字。参数 size 指定段的大小，但它以 PAGE_SIZE 的值为界，这个值是某种处理器本身页的大小（目前的 Intel 处理器是 4KB，Alpha 处理器是 8KB）。shmget 执行成功时返回段标识符，出错则返回-1。

程序清单 17.7 中的 mkshm.c 创建一块共享内存区，并且显示 shmget 返回的标识符。运行 make mkshm，使用本书提供的 makefile 文件编译这个程序。

程序清单 17.7 mkshm.c

```
/*
 * mkshm.c - Create and initialize shared memory segment
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFSZ 4096 /* Size of the segment */

int main(void)
{
    int shmid;

    if((shmid = shmget(IPC_PRIVATE, BUFSZ, 0666)) < 0) {
        perror("shmget");
        exit(EXIT_FAILURE);
    }

    printf("segment created: %d\n", shmid);
    system("ipcs -m");

    exit(EXIT_SUCCESS);
}
```

这个程序示范运行的结果如下：

```
$/mkshm
segment created: 40833

-----Shared Memory Segments-----
key          shmid  owner   perms   bytes   nattch   status
0x00000000   11375   kwall   666     4096    0
```

正如输出所显示的那样，mkshm 成功地创建了一块共享内存区。命令 `ipcs -m` 输出的倒数第二列 `nattch` 显示出已经附加了这个段（这意味着它们把段映像到它们自己的地址空间中）的进程数。注意，没有进程附加这个内存段。shmget 只创建了共享内存区；进程要把它映像到自己的地址空间（称为附加到段），必须显式地使用下一节讨论的 shmat 调用来完成。

17.4.2 附加共享内存区

在附加共享内存区之前，你都不能使用它。类似地，当你使用完毕一块共享内存区后，必须把它从你的地址空间中分离出去。使用 shmat 调用完成共享内存区的附加，而使用 shmdt 调用完成共享内存区的分离。这两个例程的原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr, int flags);
int shmdt(char *shmadr);
```

shmid 是你要附加的共享内存区的标识符。在函数 shmat 中，如果 shmadr 为 0，则内核会把段映像到调用进程的地址空间中它所选定的位置。如果 shmadr 不为 0，因为内核把段映像到它所指定的地址，显然这样做不啻为撞大运，所以总是把 shmaddr 设置为 0。flags 可以为 SHM_RDONLY，这意味着被附加的段是只读的。否则，被附加的段默认是可读写的。如果 shmat 调用成功，则返回被附加了段的地址。否则，它返回 -1 并且设置 errno 变量。

shmdt 附加在 shmaddr 的段从调用进程的地址空间中分离出去。这个地址必须是调用 shmget 返回的。

程序清单 17.8 中的第一个例子附加并分离了一块共享内存区。执行 `make atshm` 编译这个程序。

程序清单 17.8 atshm.c

```
/*
 * atshm.c - Attaching a shared memory segment
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int shmid; /* Segment ID */
    char *shmbuf; /* Address in process */

    /* Expect an segment id on the command line */
    if(argc != 2) {
```

```

    puts("USAGE: atshm <identifier>");
    exit(EXIT_FAILURE);
}
shmidx = atoi(argv[1]);

/* Attach the segment */
if((shmbuf = shmat(shmid, 0, 0)) < (char *)0) {
    perror("shmat");
    exit(EXIT_FAILURE);
}
/* Where is it attached? */
printf("segment attached at %p\n", shmbuf);

/* See, we really are attached! */
system("ipcs -m");

/* Detach */
if((shmdt(shmbuf)) < 0) {
    perror("shmdt");
    exit(EXIT_FAILURE);
}
puts("segment detached");

/* Yep, we really did detach it */
system("ipcs -m");

exit(EXIT_SUCCESS);
}

```

shmat 返回一个字符指针，所以在检查它的返回码时，atshm 把 0 强制转换为 char * 类型以避免讨厌的编译器警告消息。这个例子还使用了 ipcs 命令确认调用进程实际上确实成功地附加和分离内存区。执行 atshm，把 mkshm 返回的标识符传递给它。随后的输出演示了这一点。注意，被附加的进程数 nattch 先增加后减小。

```

$ ./atshm 11137
segment attached at 0x2aabf000

----- Shared Memory Segments -----
key          shmid  owner  perms  bytes  nattch  status
0x00000000   11137  kwall  666    4096    1
segment detached

----- Shared Memory Segments -----
key          shmid  owner  perms  bytes  nattch  status
0x00000000   11137  kwall  666    4096    0
$

```

程序清单 17.9 中的 opshm.c 显示了如何向共享内存区写入数据以及如何从共享内存区读出数据。这个程序附加一个段，向段中写入数据，然后再把缓冲写到文件 opshm.out 里。

※

通过执行 `make opshm` 使用本书提供的 `makefile` 文件编译这个程序。执行时把 `mkshm` 返回的标识符作为参数传递给它。

程序清单 17.9 opshm.c

```
/*
 * opshm.c - Reading and writing a shared memory segment
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <ctype.h>

#define BUFSZ 4096

int main(int argc, char *argv[])
{
    int shmid; /* Segment ID */
    char *shmbuf; /* Address in process */
    int fd; /* File descriptor */
    int i; /* Counter */

    /* Expect a segment id on the command line */
    if(argc != 2) {
        puts("USAGE: opshm <identifier>");
        exit(EXIT_FAILURE);
    }
    shmid = atoi(argv[1]);

    /* Attach the segment */
    if((shmbuf = shmat(shmid, 0, 0)) < (char *)0) {
        perror("shmat");
        exit(EXIT_FAILURE);
    }

    /* Size shmbuf appropriately */
    if((shmbuf = malloc(sizeof(char) * BUFSZ)) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    for(i = 0; i < BUFSZ; ++i) {
        shmbuf[i] = rand();
    }

    /* Write the segment's raw contents out to a file */
    fd = open("opshm.out", O_CREAT | O_WRONLY, 0600);
    write(fd, shmbuf, BUFSZ);
}
```

```
    free(shmbuf); /* Don't want memory leaks */  
    exit(EXIT_SUCCESS);  
}
```

这个程序执行完毕以后，你会在当前目录下发现文件 `opshm.out`。不要用 `cat` 命令查看它，因为它充满了随机产生的垃圾数据（因为 `opshm` 使用 `rand` 调用产生数据填充共享内存区）。用 `cat` 查看这个文件可能会导致终端无法使用。

17.5 消息队列

消息队列是一个消息的链接列表，消息都保存在内核中，进程通过一种和共享内存区使用的标识符同种类的标识符标识消息。为了简洁和方便，本章使用术语队列和队列 ID 分别指消息队列和消息队列标识符。如果你把一个消息添加到一个队列，队列显示出 FIFO 的特性，因为新消息被添加到队列的末尾。但是，和 FIFO 不同，队列中的消息能够以有些随意的顺序进行检索，因为，正如你所看到的那样，你可以用一个消息的类型来把消息从队列中检索出来。从这方面看，队列在编程上类似于结合数组。

所有的队列操作函数都在 `<sys/msg.h>` 中声明，但是你还必须包含 `<sys/types.h>` 和 `<sys/ipc.h>` 来访问后两个头文件中包含的类型和常量声明。要创建一个新的队列或打开一个队列，使用 `msgget` 函数。为了在队列末尾创建一个新消息，可以使用 `msgsnd` 函数。要从队列中取出一个消息，可以使用 `msgrcv` 调用。假如调用 `msgctl` 的进程是队列的创建者或者具有超级用户权限，那么 `msgctl` 能够让你控制队列的特性和删除队列。

17.5.1 创建和打开消息队列

`msgget` 函数创建一个新队列或打开一个已有的队列。它的原型如下：

```
#include <sys/types.h>  
#include <sys/ipc.h>  
#include <sys/msg.h>  
int msgget(key_t key, int flags);
```

如果调用成功，它相对应包含在 `key` 中值，返回新的或已有队列的队列 ID。如果 `key` 为 `IPC_PRIVATE`，则用底层实现生成的一个关键字的值创建一个新队列；如果没有超出系统对队列数量和所有队列中总字节数的限制，则使用 `IPC_PRIVATE` 保证了创建一个新的队列。

类似地，如果 `key` 不是 `IPC_PRIVATE`，`key` 也和一个已有队列的 `key` 不相同，而且在参数 `flags` 中设置了 `IPC_CREAT` 位，那么就创建一个新队列。否则——也就是说，如果 `key` 不是 `IPC_PRIVATE` 而且参数 `flags` 中没有设置 `IPC_CREAT` 位——`msgget` 返回和 `key` 关联的已有队列的队列 ID。如果 `msgget` 执行失败，它返回 -1 并且设置出错变量 `errno` 的值。

程序清单 17.10 中的 `mkq.c` 创建一个新的消息队列。如果你再次执行这个程序，它只是打开已有的队列而不是创建指定的队列。你可以执行 `make mkq` 使用本书提供的 `makefile`

文件编译这个程序。

程序清单 17.10 mkq.c

```
/*
 * mkq.c - Create a SysV IPC message queue
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int qid; /* The queue identifier */
    key_t key; /* The queue key */

    key = 123;

    /* Create the queue */
    if((qid = msgget(key, IPC_CREAT | 0666)) < 0) {
        perror("msgget:create");
        exit(EXIT_FAILURE);
    }
    printf("created queue id = %d\n", qid);

    /* Open the queue again */
    if((qid == msgget(key, 0)) < 0) {
        perror("msgget:open");
        exit(EXIT_FAILURE);
    }
    printf("opened queue id = %d\n", qid);

    exit(EXIT_SUCCESS);
}
```

这个程序的输出应该类似下面：

```
$ ./mkq
created queue id = 128
opened queue id =128
$
```

如果第一个 `msgget` 调用成功，`mkq` 显示新创建的队列的 ID，然后再次调用 `msgget`。如果第二次调用成功，`mkq` 也会报告。但是，第二次调用只打开已有的队列而不创建新队列。注意，第一次调用用标准八进制记法规定所有用户拥有读写权。

提示： 在创建一个 System V IPC 对象的时候，进程的 `umask` 不会修改对象的访问权限。如果没有设置访问权限，默认的模式为 0，这意味着即使是该结构的创建者也没有对它的读写权。

17.5.2 向队列中写入消息

要把一个新消息添加到队列的末尾，可以使用 `msgsnd` 函数，它的原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flags);
```

如果 `msgsnd` 执行成功返回 0，但如果执行失败则返回 -1 并且设置全局出错变量 `errno` 的值为下列值之一：

- EAGAIN
- EACCES
- EFAULT
- EIDRM
- EINTR
- EINVAL
- ENOMEM

参数 `msqid` 必须是前面调用 `msgget` 返回的队列 ID。`nbytes` 是所添加消息的字节数，消息不应该以 `null` 结尾。`ptr` 是指向 `msgbuf` 结构的指针，这个结构由消息类型和组成消息的数据字节所构成。`msgbuf` 在 `<sys/msg.h>` 中的定义如下：

```
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

这个结构声明实际只是一个模板，因为 `mtext` 必须由保存的数据来确定长度，它和传递给参数 `nbytes` 的值减去任何末尾的 `null` 字符相对应。`mtype` 可以是任何大于 0 的 `long` 型整数。调用进程还必须具有对这个队列的写权。最后，`flags` 变量不是 0 就是 `IPC_NOWAIT`。`IPC_NOWAIT` 引起的行为类似于给系统调用 `open` 传递 `O_NONBLOCK` 标志的情况：如果单个排队消息的总数或者队列以字节计的长度等于系统特定的限制，`msgsnd` 立即返回并且设置 `errno` 变量为 `EAGAIN`。因此，你不能向队列添加任何更多的消息，直到由至少一个消息被读取后为止。

如果 `flags` 为 0，而且不是最大数目的消息已被写入队列就是最大数据总字节数已被写入队列，那么 `msgsnd` 调用阻塞（不返回）直到这个条件被清除为止。要清除条件状态，必须从队列读取并删除消息（设置 `errno` 为 `EIDRM`），或者捕获到一个信号且处理函数返回（设置 `errno` 为 `EINTR`）。

提示： 可以对 `msgbuf` 结构模板进行扩展以满足你的应用程序的需要。例如，如果你要传送一个由一个整数和一个有 10 个字节的字符串组成的消息，只需按下面的方式声明 `msgbuf`：

```

struct msgbuf{
    long mtype;
    int i;
    char c[10];
};

```

msgbuf 可简单地看作是一个 long 型整数后跟消息数据，消息数据可以采用你觉得合适的格式。在本例中声明的结构的长度是 `sizeof(msgbuf)-sizeof(long)`。

程序清单 17.11 中的 `qsnd` 向一个已经存在的队列添加一条消息。这个程序唯一的命令行参数是 `mkq` 返回的队列 ID。通过运行 `make qsnd` 使用本书提供的 `makefile` 文件编译这个程序。

程序清单 17.11 `qsnd.c`

```

/*
 * qsnd.c - Send a message to previously opened queue
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define BUFSZ 512

/* Message structure */
struct msg {
    long msg_type;
    char msg_text[BUFSZ];
};

int main(int argc, char *argv[])
{
    int qid; /* The queue identifier */
    int len; /* Length of data sent */
    struct msg pmsg; /* Pointer to message structure */

    /* Expect the queue ID passed on the command line */
    if(argc != 2) {
        puts("USAGE: qsnd <queue ID>");
        exit(EXIT_FAILURE);
    }
    qid = atoi(argv[1]);

    /* Get the message to add to the queue */
    puts("Enter message to post:");
    if((fgets(&pmsg->msg_text, BUFSZ, stdin)) == NULL) {

```

```

        puts("no message to post");
        exit(EXIT_SUCCESS);
    }

    /* Associate the message with this process */
    pmsg.msg_type = getpid();
    /* Add the message to the queue */
    len = strlen(pmsg.msg_text);
    if((msgsnd(qid, &pmsg, len, 0)) < 0) {
        perror("msgsnd");
        exit(EXIT_FAILURE);
    }
    puts("message posted");
    exit(EXIT_SUCCESS);
}

```

这个程序示范运行产生的输出如下。注意程序使用 `mq` 返回的队列 ID 作为参数。

```

$./qsnd 128
Enter message to post:
Adding a message to queue ID 128.
message posted

```

命令 `qsnd 128` 产生一条提示，要求输入给队列增加的消息的内容，并且把键入的响应（以粗体表示）直接保存在早先声明的 `msg` 结构中。如果 `msgsnd` 成功完成，它就显示加入了消息。注意，`qsnd` 把消息类型设置为调用进程的 PID。这可以让你以后（用 `msgrcv`）再把这个进程加入的消息检索出来。

17.5.3 读取队列中的消息

要从队列中取出一条消息，可以使用 `msgrcv`，它的语法如下：

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flags);

```

如果执行成功，`msgrcv` 删除从队列返回的消息。它的参数和 `msgsnd` 接受的参数一样，差别在于 `msgrcv` 用消息类型和多达 `nbytes` 字节的数据填入 `ptr` 结构。另外的参数 `type` 和前面讨论的 `msg` 结构的成员 `msg_type` 相对应。`type` 的值决定了返回哪个消息，如下所述：

- 如果 `type` 是 0，返回队列中的第一条消息（最上面的一条消息）。
- 如果 `type` 大于 0，返回 `msg_type` 等于 `type` 的第一条消息。
- 如果 `type` 小于 0，返回 `msg_type` 为小于等于 `type` 绝对值的最小值的第一条消息。

`flags` 的值也控制着 `msgrcv` 的行为。如果 `flags` 中设置了 `MSG_NOERROR` 位，那么如果返回的消息比 `nbytes` 字节多，消息就会被截短到 `nbytes` 字节（但不会产生提示说消息被

截短)。否则, `msgrcv` 返回-1 表明出错并且设置 `errno` 变量的值为 `E2BIG`。消息仍旧保存在队列中。

如果 `flags` 中设置了 `IPC_NOWAIT` 位, 那么如果没有指定类型的消息, `msgrcv` 就立即返回, 并且设置 `errno` 变量为 `ENOMSG`。否则, `msgrcv` 阻塞直至出现先前描述的条件之一为止。

注意: 参数 `type` 的负值也可以用于创建一个 LIFO, 即先入后出类型的队列, 这也就是人们所熟悉的栈。在 `type` 中传递负值能够让你在队列中以逆序检索某种类型的消息。

程序清单 17.12 中的 `qrd.c` 从先前创建并部署的队列读出一条消息。供读取消息的队列作为命令行参数传递给程序。命令 `make qrd` 使用本书提供的 `makefile` 文件编译这个程序。

程序清单 17.12 `qrd.c`

```
/*
 * qrd.c - Read all message from a message queue
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFSZ 512

/* Message structure */
struct msg {
    long msg_type;
    char msg_text[BUFSZ];
};

int main(int argc, char *argv[])
{
    int qid; /* The queue identifier */
    int len; /* Length of message */
    struct msg pmsg; /* A message structure */

    /* Expect the queue ID passed on the command-line */
    if(argc != 2) {
        puts("USAGE: qrd <queue ID>");
        exit(EXIT_FAILURE);
    }
    qid = atoi(argv[1]);

    /* Retrieve and display a message from the queue */
    len = msgrcv(qid, &pmsg, BUFSZ, 0, 0);
    if(len > 0) {
        (&pmsg)->msg_text[len] = '\0';
```

```

        printf("reading queue id: %05d\n", qid);
        printf("message type: %05ld\n", (&pmsg)->msg_type);
        printf("message length: %d bytes\n", len);
        printf("message text: %s\n", (&pmsg)->msg_text);
    } else {
        perror("msgrcv");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}

```

下面是这个程序一次执行的输出结果。和先前的一样，它也使用本节早先介绍的 `mkq` 运行创建的队列 ID。类似地，它读取前面的例程 `qsnd` 加入的消息。

```

$ ./qrd 640
message type: 06360
message length: 34 bytes
message text: Adding a message to queue ID 128.
$

```

从这段代码可以看出，从消息队列读出消息要比写入消息简单，代码量也小。在这个示例程序中最特别的地方在于它取回消息队列顶部的第一条消息，因为它给参数 `type` 传递的值为 0。在这种情况下，因为已知写入消息的进程的 PID，或者可以轻易地找到这个 PID，`qrd` 能够传送值为 06360 的 `type` 并从队列取回同一消息。

17.5.4 删除消息队列

`msgctl` 函数提供了对消息队列的在一定程度上的控制功能。它的原型如下：

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);

```

一般而言，`msqid` 是一个已经存在的队列的 ID。`cmd` 可以为如下值之一：

- `IPC_RMID`——删除队列 `msqid`。
- `IPC_STAT`——用队列的 `msqid_ds` 结构填充 `buf`，并让你查看队列的内容而不会删除任何消息。因为这是一种非破坏性读操作，你可以认为它和 `msgrcv` 类似。
- `IPC_SET`——让你改变队列的 UID、GID、访问模式和队列的最大字节数。

程序清单 17.13 中的 `qctl.c` 使用 `msgctl` 调用删除一个队列，队列 ID 通过命令行传递。通过执行 `make qctl` 使用本书提供的 `makefile` 文件编译这个程序。

程序清单 17.13 qctl.h

```

/*
 * qctl.c - Remove a message queue
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int qid;

    if(argc != 2) {
        puts("USAGE: qctl <qid>");
        exit(EXIT_FAILURE);
    }
    qid = atoi(argv[1]);

    if((msgctl(qid, IPC_RMID, NULL)) < 0) {
        perror("msgctl");
        exit(EXIT_FAILURE);
    }
    printf("queue %d removed\n", qid);
    exit(EXIT_SUCCESS);
}

```

因为例程 `qrd` 也删除了先前创建的队列，所以你不能不在使用 `qctl` 之前运行 `mkq` 创建一个新队列。下面显示了一个运行的例子：

```

$ ./mkq
created queue id = 384
opened queue id = 384
$ ipcs -q

----- Message Queues -----
key          msqid   owner   perms      used-bytes   messages
0x0000007b   384      kwall   666         0             0

$ ./qctl 384
queue 384 removed
$ ipcs -q

----- Message Queues -----
key          msqid   owner   perms      used-bytes   messages
$

```

此处的程序输出表明指定的队列被删除了。mkq 创建队列。ipcs 调用确认队列被创建。通过使用 mkq 返回的队列 ID，qctl 调用 msgctl 函数并指定 IPC_RMID 标志来删除这个队列。再次运行 ipcs 证实 qctl 删除了队列。

在这几个例程中使用的 ipcs 命令是大多数 Linux 发布版本安装的 IPC 软件套件的一部分。它显示了在执行的时候系统上所有 System V IPC 结构的数目和状态。ipcrm 命令能删除在命令行指定其类型和 ID 的 IPC 结构。参考手册页面了解更多的信息。

17.6 信 号 灯

信号灯控制对共享资源的访问，就好像十字路口的交通灯控制交通流量一样。它们和迄今你所看到的其他 IPC 方式相当不同，因为它们并不能在进程间共享信息，而是同步对共享资源的访问，这些资源不能被同时访问。从这方面来看，信号灯的用法和文件或记录加锁机制非常类似，但区别在于信号灯可以用于文件以外的更多资源。本节只讨论最简单类型的信号灯：双态信号灯。一个双态信号灯只能取两个值中的一个：当资源被加锁不能被其他进程访问时为 0，而当资源解锁时为 1。

怎样使用信号灯呢？当一个进程需要访问一个受控资源，比如文件时，它首先检查信号灯的值，就好比司机查看交通灯是否为绿灯一样。如果信号灯的值 0，这相当于红灯，则资源处于使用状态，所以进程阻塞直至能够使用该资源为止（也就是说，信号灯的值变为非 0）。用 System V IPC 的术语来说，这种阻塞称为等待。如果信号灯为正值，这相当于十字路口的绿灯，则相关资源可用，所以进程减小信号灯的值，在资源上执行它的操作，然后增加信号灯的值来释放锁。

17.6.1 创建信号灯

自然，在你能增加或减小一个信号灯的值之前它必须存在才行。创建一个新的信号灯或访问一个已经存在的信号灯的函数调用是 semget，其原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int flags);
```

semget 返回和 nsems 信号灯集合相关的信号灯标识符。如果 key 为 IPC_PRIVATE，或者如果 key 还没有使用并且在 flags 中设置了 IPC_CREAT 位，则创建新的信号灯集合。和共享内存区和消息队列类似，flags 也能和权限位（八进制形式）按位进行“或”操作来为信号灯设置访问模式。但是要注意，信号灯有读取和改变的权限而不是读取和写入的权限。信号灯使用改变而不是写入的概念，因为你实际上决不会向信号灯写入数据，你只会通过增加或减小信号灯的值来改变（或者调整）它的状态。如果出错，semget 返回-1 并且设置 errno 变量。否则它返回和 key 的值相关联的信号灯标识符。

注意： System V 信号灯调用实际上是对信号灯数组或者说集合，而不是单个信

号灯进行操作。但是本章的目的在于简化讨论并且讲述实用知识，而不是全面介绍信号灯的复杂内容。不管你使用的是单个信号灯还是多个信号灯，基本方法都是一样的，但是对你来说，理解 System V 信号灯的集合形式是很重要的。POSIX IPC 制订了更简单但功能丝毫不减的信号灯接口标准。

`semop` 函数是最常用的信号灯例程。它在一个或多个由 `semget` 调用创建或访问的信号灯上执行操作。`semop` 函数的原型如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *semops, unsigned nops);
```

`semid` 是先前 `semget` 返回的信号灯标识符，它指向要操作的信号灯集合。`nops` 是 `semops` 指向的 `sembuf` 结构数组的元素个数。接下来 `sembuf` 结构的定义如下：

```
struct sembuf {
    short sem_num; /* Semaphore number */
    short sem_op;  /* The operation to perform */
    short sem_flg; /* Flags controlling the operation */
};
```

在 `sembuf` 结构中，`sem_num` 是信号灯的编号，其值从 0 到 `nsems - 1`，`sem_op` 是执行的操作，而 `sem_flg` 调整 `semop` 的行为。`sem_op` 能够为正值、负值和 0。

- 如果 `sem_op` 为正，信号灯控制的资源被释放，而且信号灯的值增加。
- 如果 `sem_op` 为负，调用进程表示它将等待直到受控资源被释放，此时信号灯的值减小而资源被调用进程加锁。
- 如果 `sem_op` 为 0，调用进程阻塞直到信号灯变为 0；如果信号灯已经是 0，调用立即返回。`sem_flg` 可以为 `IPC_NOWAIT`，其行为前面已经描述过，或者为 `SEM_UNDO`，这意味着当调用 `semop` 的进程退出后执行的操作将被撤销。

程序清单 17.14 中的 `mksem.c` 创建一个信号灯并增加它的值，因而标记这个资源已被解锁或者说资源可用。使用本书提供的 `makefile` 文件执行 `make mksem` 命令编译这个程序。

程序清单 17.14 `mksem.c`

```
/*
 * mksem.c - Create and increment a semaphore
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
```

```

int semid; /* Semaphore identifier */
int nsems = 1; /* How many semaphores to create */
int flags = 0666; /* World read-alter mode */
struct sembuf buf; /* How semop should behave */

/* Create the semaphore with world read-alter perms */
semid = semget(IPC_PRIVATE, nsems, flags);
if(semid < 0) {
    perror("semget");
    exit(EXIT_FAILURE);
}
printf("semaphore created: %d\n", semid);

/* Set up the structure for semop */
buf.sem_num = 0; /* A single semaphore */
buf.sem_op = 1; /* Increment the semaphore */
buf.sem_flg = IPC_NOWAIT; /* Don't block */

if((semop(semid, &buf, nsems)) < 0) {
    perror("semop");
    exit(EXIT_FAILURE);
}
system("ipcs -s");
exit(EXIT_SUCCESS);
}

```

运行一次 `mksem` 其输出结果如下。输出显示的标识符的值可能和在你的系统上运行看到的不同。

```

$ ./mksem
semaphore created: 0

----- Semaphore Arrays -----
key          semid  owner    perms    nsems    status
0x00000000    0      kwall    666      1

```

使用 `IPC_PRIVATE` 的例子保证了按要求创建信号灯，然后显示 `semget` 的返回值，即信号灯的标识符。调用 `semop` 恰当地对信号灯进行初始化：既然只创建了一个信号灯，因此 `sem_num` 为 0。因为想像中的资源没有处于使用状态（更精确地说，信号灯没有在程序上和任何特定资源相关联），`mksem` 把它的值初始化为 1，这等价于解锁。在这里的情况下不要求阻塞行为，信号灯的 `sem_flg` 设置为 `IPC_NOWAIT`，所以调用立即返回。最后，这个例子使用 `system` 函数调用用户级的 `ipcs` 命令以确定所要求的 IPC 结构确实存在。

17.6.2 控制和删除信号灯

你已经看到了 `msgctl` 和 `shmctl` 函数，这两个函数控制消息队列和共享内存区。正如你所期望的那样，用于信号灯的等价函数是 `semctl`，它的原型如下：

```
#include <sys/types.h>
```

```
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, union semun arg);
```

`semid` 标识出要操作的信号灯集合，而 `semnum` 指出感兴趣的特定信号灯。本书忽略一个集合中含有多个信号灯的情况，所以 `semnum`（实际上是对信号灯数组的索引）总是 0。参数 `cmd` 可能的取值如表 17.1 所示。

表 17.1 `semctl` 调用中 `cmd` 的取值

命令	描述
GETVAL	返回信号灯当前的状态（加锁或解锁）
SETVAL	设置信号灯当前的状态为 <code>arg.val</code> （参数 <code>semun</code> 在本书的其他地方讨论）
GETPID	返回上次调用 <code>semop</code> 的进程的 PID
GETNCNT	导致 <code>semctl</code> 的返回值是正在等待信号灯的值增加的进程数——也就是说，在信号灯上等待的进程数
GETZCNT	导致 <code>semctl</code> 的返回值是等待信号灯的值 0 的进程数
GETALL	返回和 <code>semid</code> 相关联的集合中所有信号灯的值
SETALL	设置和 <code>semid</code> 相关联的集合中所有信号灯的值保存在 <code>arg.array</code> 中的值
IPC_RMID	删除带有 <code>semid</code> 的信号灯
IPC_SET	在信号灯上设置模式（权限位）
IPC_STAT	每个信号灯都有一个数据结构 <code>semid_ds</code> ，这个数据结构完整地描述它的配置和行为。IPC_STAT 把这些配置信息复制到 <code>semun</code> 结构的成员 <code>arg.buf</code> 中

如果 `semctl` 调用执行失败，则返回 -1 并且恰当地设置变量 `errno` 的值。否则，它根据 `cmd` 的值返回 GETNCNT、GETPID、GETVAL 或 GETZCNT 中的一个整数值。

正如你能总结出来的那样，参数 `semun` 在 `semctl` 例程中扮演了一个重要角色。你必须在自己的代码中按照下面的模板定义它：

```
union semun {
    int val; /* Value for SETVAL */
    struct semid_ds *buf; /* IPC_STAT's buffer */
    unsigned short int *array; /* GETALL and SETALL's buffer */
};
```

到现在为止，你应该开始理解认为 System V 信号灯太复杂的抱怨了。虽然困难，但程序清单 17.15 中的程序 `sctl.c` 还是使用 `semctl` 调用从系统中删除了一个信号灯。你要先执行 `mksem` 创建一个信号灯，然后再把该信号灯的标识符作为参数传递给 `sctl`。执行 `make sctl` 编译这个程序。运行这个程序时要把 `mksem` 返回的信号灯 ID 作为参数传递给它。

程序清单 17.15 `sctl.c`

```
/*
 * sctl.c - Manipulate and delete a semaphore
 */
#include <sys/types.h>
#include <sys/ipc.h>
```

```

#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int semid; /* Semaphore identifier */

    if(argc != 2) {
        puts("USAGE: sctl <semaphore id>");
        exit(EXIT_FAILURE);
    }
    semid = atoi(argv[1]);

    /* Remove the semaphore */
    if((semctl(semid, 0, IPC_RMID)) < 0) {
        perror("semctl IPC_RMID");
        exit(EXIT_FAILURE);
    } else {
        puts("semaphore removed");
        system("ipcs -s");
    }

    exit(EXIT_SUCCESS);
}

```

下面显示了在我的系统上 sctl 的输出：

```

$ ./sctl 0
semaphore removed

----- Semaphore Arrays -----
key      semid      owner      perms      nsems      status

```

sctl 的代码只是试图删除从命令行传入的标识符所标识出的信号灯。它还使用 system 调用执行 ipcs -s 命令以确认信号灯被删除了。

17.7 小 结

本章介绍了 Linux 上进程间通信机制的多种形式：管道和 FIFO、System V IPC 共享内存、信号灯和信息队列。虽然共享内存 IPC 比管道和 FIFO 复杂，但是它的功能更强大也更灵活，常常在更大型更复杂的应用中使用，比如类似 Informix 和 Oracle 这样的关系数据库管理系统（relational database management systems, RDBMS）。消息队列能够让应用程序在无关进程间传递消息方面有很大的施展空间。最后，信号灯是一种对多种系统资源的访问进行控制的简便途径。

第 18 章 守护进程

本章向你展示如何编写守护进程程序（daemon，读作 demon）。守护进程是一个后台进程，它无需用户输入就能运行而且常常提供某种服务，不是对整个系统就是对用户程序提供服务。常见的守护进程程序包括系统日志进程、Web 服务器、邮件服务器和数据库服务器。

18.1 理解守护进程

典型地，守护进程在系统改变运行级时启动（并停止），也就是说它们一般在系统自举时启动，而且，除非强行中止它们，否则在系统关机之前一直保持运行。因为一个守护进程不能够控制终端，所以任何输出，无论是向标准出错设备 `stderr` 还是向标准输出设备 `stdout` 的输出都需做特别处理。

守护进程还有几个有别于类似 `ls` 和 `cat` 那样的普通程序的特性。首先，守护进程经常以超级用户权限运行，因为它们要使用有特权的端口（端口号从 1~1024）或者因为它们要访问某些特权资源，比如系统日志。因为它们是非交互式程序，所以它们没有控制终端；也就是说，它们不需要用户的输入。守护进程通常作为进程组和会话的领导进程。它们经常起到进程组和会话的服务进程的作用。最后，一个守护进程的父进程是 `init` 进程，它的 `PID` 为 1。这是因为它真正的父进程在 `fork` 出子进程后就先于子进程执行 `exit` 退出了，因此它是一个由 `init` 继承的孤儿进程。

18.2 创建守护进程

虽然守护进程看上去对编程来说似乎既神秘又困难，但是如果你牢记几条规则而且知道要调用的关键函数，那么编程工作就的确非常简单了。幸运的是，你只需要学习一种新的函数调用，因为你已经在前面的章节中看到过其他函数了。但是一个守护进程的出错处理带来了特别的困难，并且要求程序使用系统的日志工具 `syslog` 向系统日志（通常是文件 `/var/log/messages`）发送消息。本章 18.2.2 节“出错处理”专门讨论这个话题。

只要简单的几步就能创建出一个工作良好并和系统协调良好的守护进程。

首先，执行 `fork` 然后让父进程退出。和多数程序一样，一个守护进程是从 `shell` 脚本或命令行启动的。但是，守护进程和应用程序不一样，因为它们不是交互式的——它们在后台运行因而没有控制终端。父进程按照第一步 `fork` 子进程并退出后就消除了控制终端。这产生了完美的效果。守护进程既不需要从标准输入设备读取信息，也不需要向标准输出设备或标准出错设备写入信息，所以它们只需要终端接口的寿命能够保证启动它们就可以了。

第二步在子进程中使用 `setsid` 调用创建新的会话。调用 `setsid` 完成以下几项工作：

- 如果调用进程不是一个进程组的领导进程，它就创建一个新会话，让调用进程成为新会话的会话领导。
- 它让调用进程成为新进程组的进程组领导。
- 它把进程组 ID (PGID) 和会话 ID (SID) 设置为调用进程的进程 ID (PID)。
- 它消除新进程和任何控制终端的关联。

下一步让根目录成为子进程的当前工作目录。这是必要步骤，因为任何进程如果它的当前目录是在一个被安装的文件系统上，那么就会妨碍这个文件系统被卸载。通常只是希望如此，但是如果系统出于某种原因要进入单用户模式，运行在被安装的系统上的守护进程在最好的情况下会成为超级用户讨厌的麻烦（因为他必须找出引起问题的进程并且杀死它），而在紧急情况下则会成为对系统一致性的实在威胁（因为它不让出错磁盘上的文件系统卸载）。让“/”成为一个守护进程的工作目录是避免出现上述两种可能的问题的安全方法。

接下来，设置进程的 `umask` 为 0。为了避免守护进程继承的 `umask` 受到创建文件和目录操作的干扰，这一步也是必要的。考虑下面的情形：一个守护进程继承了 `umask 055`，它屏蔽掉了 `group` 和 `other` 的读权和执行权。如果守护进程接着创建了一个文件，如数据文件，创建出的文件对于用户来说可读、可写并且可执行，但只对 `group` 和 `other` 可写。重新设置守护进程的 `umask` 为 0 避免了这种情况。当创建文件时它还给予守护进程更大的灵活性，`umask` 为 0 时，守护进程能够精确地设置所要求的权限，而不是配置为系统默认值。

最后，关闭子进程继承的任何不必要的文件描述符。这只是符合常识的一步。对于子进程来说，没有理由保持从父进程继承来的打开的文件描述符。要关闭的潜在的文件描述符至少应包括 `stdin`、`stdout` 和 `stderr`。其他打开的文件描述符，如那些引用配置或数据文件的描述符也要关闭。这一步取决于具体的守护进程的需要和要求，所以很难更精确地说明规则。

下面的清单总结了编写守护进程时遵循的步骤：

1. 在父进程中执行 `fork` 并执行 `exit` 退出。
2. 在子进程中调用 `setsid`。
3. 让根目录“/”成为子进程的工作目录。
4. 把子进程的 `umask` 变为 0。
5. 关闭任何不需要的文件描述符。

18.2.1 函数调用

为了满足第一条的要求，调用 `fork` 产生一个子进程，然后让父进程调用 `exit` 退出。为了消除控制终端，调用 `setsid` 创建一个新会话，该函数的原型如下：

```
#include <unistd.h>
pid_t setsid(void);
```


`setsid` 创建一个新会话和一个新进程组。然后守护进程成为新会话的会话领导，以及新进程组的进程组领导。`setsid` 调用还保证新会话没有控制终端。但是，如果调用进程已经是一个进程组的领导进程，`setsid` 调用将执行失败。`setsid` 执行成功时返回新会话的 ID。失败时返回 -1 并且设置 `errno` 变量。

为了改变工作目录，使用 `chdir` 调用。`umask` 调用把守护进程的 `umask` 设置为 0。如前所述，这样做取消了任何继承的 `umask`，它们能够潜在地干扰创建文件或目录。调用 `close` 关闭不需要的文件描述符。

程序清单 18.1 显示了一个简单的守护进程的代码。`lpupdate` 创建一个新的日志文件（或者附加到已经存在的文件后面）`/var/log/lpupdated.log`，每分钟向其中写入一个时间戳。为了能让 `lpupdated` 工作，必须由超级用户启动它。执行 `make lpupdated` 使用本书提供的 `makefile` 文件编译这个程序。

程序清单 18.1 `lpupdated.c`

```
/*
 * lpupdated.c - Simple timestamping daemon
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <time.h>
#include <syslog.h>

int main(void)
{
    pid_t pid, sid;
    time_t timebuf;
    int fd, len;

    pid = fork();
    if(pid < 0) {
        perror(fork);
        exit(EXIT_FAILURE);
    }
    if(pid > 0)
        /* In the parent, let's bail */
        exit(EXIT_SUCCESS);

    /* In the child... */
    /* First, start a new session */
    if((sid = setsid()) < 0) {
        perror("setsid");
    }
}
```

```
        exit(EXIT_FAILURE);
    }

    /* Next, make / the current directory */
    if((chdir("/") < 0) {
        perror("chdir");
        exit(EXIT_FAILURE);
    }

    /* Reset the file mode */
    umask(0);

    /* Close unneeded file descriptors */
    close(STDIN_FILENO);
    close(STDOUT_FILENO);
    close(STDERR_FILENO);

    /* Finally, do our work */
    len = strlen(ctime(&timebuf));
    while(1) {
        char *buf = malloc(sizeof(char) * (len + 1));

        if(buf == NULL) {
            perror("malloc");
            exit(EXIT_FAILURE);
        }

        if((fd = open("/var/log/lpupdated.log",
                     O_CREAT | O_WRONLY | O_APPEND, 0600)) < 0) {
            perror("open");
            exit(EXIT_FAILURE);
        }

        time(&timebuf);
        strncpy(buf, ctime(&timebuf), len + 1);
        write(fd, buf, len + 1);
        close(fd);
        sleep(60);
    }

    exit(EXIT_SUCCESS);
}
```

要启动这个程序，先变为超级用户，并在你编译这个程序的目录下执行命令 `lpupdated`。

`lpupdated` 使用的系统调用 `open` 和 `write` 会让你回忆起 Linux 提供了标准库的流 I/O 函数 `fopen` 和 `fwrite` 的一种替代方法。请记住，你必须以超级用户身份运行这个程序，因为普通用户没有对 `/var/log` 目录的写权限。如果你试图以非超级用户身份运行这个程序，则什么都不会发生。守护进程在打开它的日志文件失败后就简单地终止运行。几分钟之后，由守护进程维护的日志文件 `/var/log/lpupdated.log` 的内容应该和下面的类似：

```
$ su -c "tail -f /var/log/lpupdated.log"
Password:
Web Aug 23 22:02:25 2000
Web Aug 23 22:03:25 2000
Web Aug 23 22:04:25 2000
Web Aug 23 22:05:25 2000
Web Aug 23 22:06:25 2000
Web Aug 23 22:07:25 2000
Web Aug 23 22:08:25 2000
```

注意，lpupdated 在调用 setsid 后停止向 stderr 写入出错消息。子进程不再有控制终端，所以输出没有地方可去。无限循环完成程序的工作：打开日志文件、输出时间戳、关闭日志文件，然后睡眠 60 秒钟。要杀死这个程序，可变为超级用户，取得 lpupdated 的 PID 然后发出 kill { lpupdated 的 PID } 命令。

18.2.2 出错处理

一旦守护进程调用 setsid，它就不再有控制终端，所以也就无处发送正常情况下应该发往 stdout 或 stderr 的输出（比如出错消息）。幸运的是，用于此项目的标准工具是源自于 BSD 的 syslog 服务，由系统日志守护进程 syslogd 提供这一服务。

相关的接口在 <syslog.h> 中定义。API 相当简单。openlog 打开日志，syslog 向其中写入消息，而 closelog 关闭日志。函数原型如下：

```
#include <syslog.h>
void openlog(char *ident, int option, int facility);
void closelog(void);
void syslog(int priority, char *format, ...);
```

openlog 发起到系统日志服务器的一个连接。ident 是要向每个消息加入的字符串，典型地应该设置为程序的名称。参数 option 是下面列出的一个或多个值的逻辑“或”：

- LOG_CONS 如果系统日志服务器不能使用，则写入控制台。
- LOG_NDELAY 立即打开连接。正常情况下，直到发送第一条消息时才打连接。
- LOG_PERROR 打印输出到 stderr。
- LOG_PID 在每条消息中包含进程的 PID。

facility 指定程序发送消息的类型，它可以是表 18.1 中列出的值之一：

表 18.1 系统日志服务器的 facility 值

Facility	描述
LOG_AUTHPRIV	安全/授权消息
LOG_CRON	时钟守护进程；cron 和 at
LOG_DAEMON	其他系统守护进程
LOG_KERN	内核消息
LOG_LOCAL[0-7]	为本地使用而保留
LOG_LPR	行打印机子系统

(续表)

Facility	描述
LOG_MAIL	邮件子系统
LOG_NEWS	Usenet 新闻组子系统
LOG_SYSLOG	syslogd 产生的消息
LOG_USER	默认
LOG_UUCP	UUCP

priority 指定消息的重要性。表 18.2 列出了它的可能值。

表 18.2 系统日志服务器的 priority 值

priority	描述
LOG_EMERG	系统不能使用
LOG_ALERT	立即采取措施
LOG_CRIT	紧急条件
LOG_ERR	出错条件
LOG_WARNING	警告条件
LOG_NOTICE	正常但重大条件
LOG_INFO	信息消息
LOG_DEBUG	调试信息

严格地说, 使用 openlog 和 closelog 是可选的, 因为函数 syslog 能在首次调用它的时候自动打开日志文件。

程序清单 18.2 用系统日志服务器重写了 lpdated 程序。这个版本在本章的源代码目录下。执行 make lpdated 使用本书提供的 makefile 文件编译这个程序。

程序清单 18.2 修订版的 lpdated.c

```

/*
 * lpdated.c - Simple timestamping daemon
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <time.h>
#include <syslog.h>

int main(void)
{
    pid_t pid, sid;
    time_t timebuf;

```

```
int fd, len;

pid = fork();
if(pid < 0) {
    syslog(LOG_ERR, "%s\n", perror);
    exit(EXIT_FAILURE);
}
if(pid > 0)
    /* In the parent, let's bail */
    exit(EXIT_SUCCESS);

/* In the child... */
/* Open the system log */
openlog("lpudated", LOG_PID, LOG_DAEMON);

/* First, start a new session */
if((sid = setsid()) < 0) {
    syslog(LOG_ERR, "%s\n", "setsid");
    exit(EXIT_FAILURE);
}

/* Next, make / the current directory */
if((chdir("/")) < 0) {
    syslog(LOG_ERR, "%s\n", "chdir");
    exit(EXIT_FAILURE);
}

/* Reset the file mode */
umask(0);

/* Close unneeded file descriptors */
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);

/* Finally, do our work */
len = strlen(ctime(&timebuf));
while(1) {
    char *buf = malloc(sizeof(char) * (len + 1));

    if(buf == NULL) {
        syslog(LOG_ERR, "malloc");
        exit(EXIT_FAILURE);
    }

    if((fd = open("/var/log/lpudated.log",
        O_CREAT | O_WRONLY | O_APPEND, 0600)) < 0) {
        syslog(LOG_ERR, "open");
        exit(EXIT_FAILURE);
    }

    time(&timebuf);
    strncpy(buf, ctime(&timebuf), len + 1);
```

```

        write(fd, buf, len + 1);
        close(fd);
        sleep(60);
    }

    /* Close the system log and scram */
    closelog();
    exit(EXIT_SUCCESS);
}

```

在加入 `syslog` 的日志功能之后，如果你试着以普通用户身份执行这个程序，`lpudated` 会在系统日志（在 Linux 系统上通常是 `/var/log/messages` 文件）中写入一条类似下面的消息：

```
Aug 23 22:21:26 hoser lpudated[10026]: open
```

这个日志项表明在指定的日期和时间里，在名为 `hoser` 的主机上，一个名为 `lpudated`、PID 为 10026 的程序向系统日志写入了文字 `open`。参照 `lpudated.c` 源代码，你能看到错误出现在什么位置。

即使时间戳本身仍然被写入到 `/var/log/lpudated.log` 中，但 `lpudated` 产生的所有出错消息都记录到系统日志中。

18.3 和守护进程通信

要和一个进程通信，你要向它发送信号让它以某种方式响应。例如，强行要求一个守护进程重新读取它的配置文件。这样做最常用的方法是向守护进程发送一个 `SIGHUP` 信号（Apache HTTP 服务器和 Sendmail 邮件服务器都把 `SIGHUP` 消息解释为重新读取它们的配置文件的消息）。另一个常见的需求是，在不强行要求守护进程重新读取配置文件的情况下改变守护进程的行为。本节的示例程序修改 `lpudated`，让它读取一个配置文件并响应 `SIGHUP` 信号。但是，首先你要学会怎样读取一个配置文件和怎样使用一个配置文件来控制守护进程的行为。

18.3.1 读取配置文件

第一个任务是教会 `lpudated` 如何读取一个配置文件。出于演示的目的，配置文件 `/etc/lpudated.conf` 只有少于 256 字符的一行文本。实际上，正如你很快就会看到的那样，`/etc/lpudated.conf` 能够包含和你所希望的任意行数一样多的文本，但是 `lpudated` 只读取前 255 个字符。程序清单 18.3 显示了修改过的程序，其名称改为 `lpudated-rc`。执行 `make lpudated-rc` 使用本书提供的 `makefile` 文件编译这个程序。

程序清单 18.3 `lpudated-rc.c`

```

/*
 * lpudated-rc.c - Simple timestamping daemon that reads a
 *                 config file in /etc
 */

```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <time.h>
#include <syslog.h>

#define RCFILE "/etc/lpupdated.conf" /* The config file */
#define BUFLen 256 /* Length of buffer reads */

int main(void)
{
    pid_t pid, sid;
    time_t timebuf;
    int fd, rcfd, len;

    pid = fork();
    if(pid < 0) {
        syslog(LOG_ERR, "%s\n", perror);
        exit(EXIT_FAILURE);
    }
    if(pid > 0)
        /* In the parent, let's bail */
        exit(EXIT_SUCCESS);

    /* In the child */
    /* Open the system log */
    openlog("lpupdated", LOG_PID, LOG_DAEMON);

    /* Read the config file */
    if((rcfd = open(RCFILE, O_RDONLY)) < 0) {
        syslog(LOG_ERR, "%s\n", "error opening RCFILE\n");
        exit(EXIT_FAILURE);
    } else {
        /* Read the config file */
        char rcbuf[BUFLen]; /* Hold the value read */
        int len; /* The length of the buffer */

        if((len = read(rcfd, rcbuf, BUFLen)) < 0) {
            syslog(LOG_ERR, "%s\n", "error reading RCFILE\n");
            exit(EXIT_FAILURE);
        }
        rcbuf[len] = '\0';

        /* Close the config file */
        if(close(rcfd) < 0) {
```

```

    syslog(LOG_ERR, "%s\n", "error closing RCFILE\n");
    exit(EXIT_FAILURE);
}

/* write the read value out to the system log */
syslog(LOG_INFO, "%s\n", rcbuf);
}

/* First, start a new session */
if((sid = setsid()) < 0) {
    syslog(LOG_ERR, "%s\n", "setsid");
    exit(EXIT_FAILURE);
}

/* Next, make / the current directory */
if((chdir("/") < 0) {
    syslog(LOG_ERR, "%s\n", "chdir");
    exit(EXIT_FAILURE);
}

/* Reset the file mode */
umask(0);

/* Close unneeded file descriptors */
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);

/* Finally, do our work */
len = strlen(ctime(&timebuf));
while(1) {
    char *buf = malloc(sizeof(char) * (len + 1));

    if(buf == NULL) {
        syslog(LOG_ERR, "malloc");
        exit(EXIT_FAILURE);
    }

    if((fd = open("/var/log/lpupdated.log",
        O_CREAT | O_WRONLY | O_APPEND, 0600)) < 0) {
        syslog(LOG_ERR, "open");
        exit(EXIT_FAILURE);
    }

    time(&timebuf);
    strncpy(buf, ctime(&timebuf), len + 1);
    write(fd, buf, len + 1);
    close(fd);
    sleep(60);
}

/* Close the system log and scram */
closelog();

```



```
    exit(EXIT_SUCCESS);
}
```

从对前面章节的回忆可知，守护进程必须由超级用户启动。加阴影的#define 语句设置了配置文件的名字 (/etc/lpupdated.conf) 和用于 read 语句的宏。第二段阴影部分显示了为读取配置文件而加入的代码。理想情况下，它应该被分成一个或几个函数来让程序具有更好的粒度。正如你所看到的那样，lpupdated-rc.c 定义了本地变量保存从配置文件读入的文本以及读入的字符数。除了严格的出错检查之外，添加的代码段没有特别显著的地方，它只是简单地打开配置文件、读出保存在其中的信息、关闭文件并且把读出的文本写入系统日志来核实 lpupdated-rc 确实成功地读取了文件。接着，它继续执行它的正常功能，每分钟把时间戳写入/var/log/lpupdated.log 文件。

注意： 在读取配置文件一段代码中声明的变量 len 和在外部声明的变量 len 不冲突。虽然通常认为以这种方式重复使用变量名是比较糟糕的形式，但 ANSI C 标准却让外部声明的变量 len 被内部声明的变量 len 所屏蔽。

配置文件/etc/lpupdated.conf 包含这样一行：

```
THIS IS THE ENTRY IN /etc/lpupdated.conf
```

出现在系统日志中文本应该和下面类似：

```
Sep 15 19:15:20 hoser lpupdated[2518]: THIS IS THE ENTRY IN
/etc/lpupdated.conf
```

18.3.2 向守护进程加入信号处理功能

一旦守护进程知道怎样读取一个配置文件，下一课就是让它能够处理信号。在它的下一个实例 lpupdated-sig 中，lpupdated 学会了怎样响应 SIGHUP，该信号让它重新读取它的配置文件。使用 make lpupdated-sig 编译这个程序。

程序清单 18.4 lpupdated-sig.c

```
/*
 * lpupdated-sig.c - Simple timestamping daemon that reads a
 *                  config file in /etc and handles signals
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <time.h>
#include <syslog.h>
#include <signal.h>
```

```

#define RCFILE "/etc/lpupdated.conf" /* The config file */
#define BUFLLEN 256 /* Length of buffer reads */
#define NORMAL 1
#define ROT13 2

/* Read the config file */
int read_config(int rcfd, int *o_style);

int main(void)
{
    pid_t pid, sid;
    time_t timebuf;
    int rcfd, fd, len, o_style;

    pid = fork();
    if(pid < 0) {
        syslog(LOG_ERR, "%s\n", perror);
        exit(EXIT_FAILURE);
    }
    if(pid > 0)
        exit(EXIT_SUCCESS);

    openlog("lpupdated", LOG_PID, LOG_DAEMON);

    /* Read the config file */
    if((rcfd = open(RCFILE, O_RDONLY)) < 0) {
        syslog(LOG_ERR, "error opening %s\n", RCFILE);
        exit(EXIT_FAILURE);
    } else {
        if((read_config(rcfd, &o_style)) < 0) {
            syslog(LOG_ERR, "error reading %s\n", RCFILE);
            exit(EXIT_FAILURE);
        }
        if(close(rcfd) < 0) {
            syslog(LOG_ERR, "error closing %s\n", RCFILE);
            exit(EXIT_FAILURE);
        }
    }

    /* First, start a new session */
    if((sid = setsid()) < 0) {
        syslog(LOG_ERR, "%s\n", "setsid");
        exit(EXIT_FAILURE);
    }

    /* Next, make / the current directory */
    if((chdir("/")) < 0) {
        syslog(LOG_ERR, "%s\n", "chdir");
        exit(EXIT_FAILURE);
    }
}

```

```

/* Reset the file mode */
umask(0);

/* Close unneeded file descriptors */
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);

/* Finally, do our work */
len = strlen(ctime(&timebuf));
while(1) {
    char *buf = malloc(sizeof(char) * (len + 1));
    sigset_t sigset;
    struct sigaction action;
    int i;
    if(buf == NULL) {
        syslog(LOG_ERR, "malloc");
        exit(EXIT_FAILURE);
    }
    if((fd = open("/var/log/lpupdated.log",
        O_CREAT | O_WRONLY | O_APPEND, 0600)) < 0) {
        syslog(LOG_ERR, "open");
        exit(EXIT_FAILURE);
    }
    sigemptyset(&sigset); /* Initialize signal set */
    sigaddset(&sigset, SIGHUP); /* Add SIGHUP to it */
    sigprocmask(SIG_BLOCK, &sigset, NULL); /* Block SIGHUP */
    time(&timebuf);
    strncpy(buf, ctime(&timebuf), len + 1);

    if(o_style == ROT13) {
        for(i = 0; i <= len; ++i)
            if((buf[i] >= 'A' && buf[i] <= 'M') ||
                (buf[i] >= 'a' && buf[i] <= 'm'))
                buf[i] += 13;
            else if((buf[i] >= 'N' && buf[i] <= 'Z') ||
                (buf[i] >= 'n' && buf[i] <= 'z'))
                buf[i] -= 13;
    }

    write(fd, buf, len + 1);
    close(fd);

    sigpending(&sigset); /* Check for pending signals */
    if(sigismember(&sigset, SIGHUP)) {
        syslog(LOG_INFO, "received SIGHUP\n");
        /* Ignore SIGHUP */
        sigemptyset(&action.sa_mask);
        action.sa_handler = SIG_IGN;
    }
}

```

```

sigaction(SIGHUP, &action, NULL);
/* Reread the configuration file */
if((rcfd = open(RCFILE, O_RDONLY)) < 0) {
    syslog(LOG_ERR, "error opening %s\n", RCFILE);
    exit(EXIT_FAILURE);
} else {
    if((read_config(rcfd, &o_style)) < 0) {
        syslog(LOG_ERR, "error reading %s\n", RCFILE);
        exit(EXIT_FAILURE);
    }
    if(close(rcfd) < 0) {
        syslog(LOG_ERR, "error closing %s\n", RCFILE);
        exit(EXIT_FAILURE);
    }
}
/* Unblock SIGHUP */
sigprocmask(SIG_UNBLOCK, &sigset, NULL);

sleep(60);
}
/* Close the system log and scram */
closelog();
exit(EXIT_SUCCESS);
}

int read_config(int rcfd, int *o_style)
{
    char rdbuf[BUFLen]; /* Hold the value read */
    int len; /* The length of the buffer */
    /* Read the config file */
    if((len = read(rcfd, rdbuf, BUFLen)) < 0) {
        syslog(LOG_ERR, "error reading %s\n", RCFILE);
        return 1;
    }
    rdbuf[len] = '\0';
    /* Determine what we read */
    if(0 == strncmp(rdbuf, "rot13", 5)) { /* Want rot13 output */
        *o_style = ROT13;
        syslog(LOG_INFO, "output style is rot13\n");
    } else if(0 == strncmp(rdbuf, "normal", 6)) { /* Want normal output */
        *o_style = NORMAL;
        syslog(LOG_INFO, "output style is normal\n");
    } else { /* Invalid value in config file */
        syslog(LOG_ERR, "invalid output style %d\n", *o_style);
        return 2;
    }
}

```

```

    }
    return 0;
}

```

正如你所看到的那样，从 `lpudated-rc` 到 `lpudated-sig` 变动很大。改变的部分同样也加了阴影。首先，正如对 `lpudated-rc` 的建议所指出的那样，读取配置文件的代码被分成了一个函数 `read_config`。它也在分析读取内容上做了很大扩展，而且设置了一个新变量 `o_style`。一直作为该程序焦点的 `while` 循环有了提供某些信号处理功能的额外代码。`sigset` 变量让 `lpudated-sig` 能够设立一个阻塞信号 (`SIGHUP`)，而 `action` 变量让程序能够修改信号的部署，此时改为 `SIG_IGN` (忽略)，所以它决不会被递送。但是，因为 `SIGHUP` 被阻塞，`lpudated-sig` 能够使用 `sigpending` 调用来检查是否出现 `SIGHUP` 信号。如果是，它就重新读取配置文件。

提示： 如果你对信号不熟悉，重新阅读第 13 章，快速复习一下。

这个练习的全部内容就是使用信号来改变守护进程的行为。`read_config` 函数在 `/etc/lpudated.conf` 中查找两个值中的一个：`rot13` 或 `normal`。如果值为 `normal`，则守护进程的时间戳输出和正常情况下一样。但是如果值为 `rot13`，下面的代码块会把每个字符在字母表中向前或者向后循环移动 13 个位置，用旋转后的值代替实际值。只有字母字符才旋转。数字则被忽略：

```

for(i = 0; i <= len; ++i)
    if((buf[i] >= 'A' && buf[i] <= 'M') || (buf[i] >= 'a' && buf[i]
        <= 'm'))
        buf[i] += 13;
    else if ((buf[i] >= 'N' && buf[i] <= 'Z') ||
        (buf[i] >= 'n' && buf[i] <= 'z'))
        buf[i] -= 13;
}

```

`rot13` 是称为凯萨密码的一类加密方法的一种非常简单的形式，这种加密方法根据公式处理文本，能够被轻易解码。例如，当用 `rot13` 加密后，单词 `program` 就变成 `cebtenz`。当然，这不是一个有用的函数，但是它能让你非常容易地看出程序是怎样工作的。

首先，确保 `/etc/lpudated.conf` 中出现值 `normal`。然后，打开两个终端窗口，分别用 `tail -f /var/log/messages` 和 `tail -f /var/log/lpudated.log` 命令查看系统日志和 `lpudated` 的日志 (`/var/log/lpudated.log`)。输出应该和下面类似：

```

$ tail -f /var/log/messages
Sep 19 16:16:56 hoser -- Mark --
Sep 19 16:36:56 hoser -- Mark --
Sep 19 16:37:00 hoser named[442]: Cleaned cache of 10 RRsets
Sep 19 16:37:00 hoser named[442]: USAGE 969403020 968985419
CPU = 3.38u/2.22s CHILDCPU=0u/0s

$ tail -f /var/log/lpudated.log
Tue Sep 19 22:53:06 2000

```

```
Tue Sep 19 22:54:06 2000
Tue Sep 19 22:55:06 2000
Tue Sep 19 22:56:06 2000
```

在第三个窗口中，启动 `lpupdated-sig`。在消息窗口中，你应该能看到非常类似下面的输出：

```
Sep 19 23:07:25 hoser lpupdated[15672]: output style is normal
```

没有什么比上述内容更直观了。`lpupdated-sig` 指出它将产生正常的输出，你可以通过查看 `lpupdated.log` 窗口来进行确认，你应该看到在这个窗口的底部出现类似下面的输出：

```
Tue Sep 19 23:07:25 2000
Tue Sep 19 23:08:25 2000
Tue Sep 19 23:09:25 2000
```

现在，编辑 `/etc/lpupdated.conf`，把值 `normal` 改为 `rot13`，然后给 `lpupdated-sig` 发送一个 `SIGHUP` 信号（使用 `kill -HUP <lpupdated-sig 的 PID>`）。在一分钟以后（`sleep(60)` 语句让这个程序睡眠 60 秒钟），系统日志会提示 `lpupdated` 现在将使用 `rot13` 风格的输出：

```
Sep 19 23:14:25 hoser lpupdated[15672]: received SIGHUP
Sep 19 23:14:25 hoser lpupdated[15672]: output style is rot13
```

在大约一分钟以后，`lpupdated.log` 窗口应该开始显示执行了 `rot13` 加密的输出。

```
Tue Sep 19 23:13:25 2000
Tue Sep 19 23:14:25 2000
Ghr Frc 19 23:15:25 2000
Ghr Frc 19 23:16:25 2000
```

当然，你的日志项看上去会略有不同。不要忘记在你做完这个试验后停止运行守护进程。

18.4 小 结

本章讨论创建守护进程的知识，守护进程是一直在运行并且通常提供某种服务的后台程序。本章向你演示了一个简单的、没有通信能力的守护进程，还介绍了怎样通过让守护进程支持信号以及提供控制其行为的配置文件，从而让守护进程更聪明一些。本章结束了第 3 部分的内容，这部分集中介绍进程和同步。你将继续阅读第 4 部分“网络编程”。

第4部分 网络编程

第19章 TCP/IP 和套接口编程

TCP 和 UDP 是传输层协议。TCP 是保证传输的面向连接的协议，而 UDP 是无连接协议，不能保证消息传送到目的地。本章介绍 TCP 及其套接口编程；UDP 及其套接口编程将在第 20 章“UDP：用户数据报协议”中讨论。

使用 TCP 或 UDP 的套接口编程是底层的技术。替代它们的技术有 RPC、RMI 和 CORBA，它们提供了在不同类型的计算机之间自动转换原始数据类型的功能以及其他高层功能。Linux 和因特网的本质就是使用普通的标准的协议和工具，我不得不承认自己对采用简单的基于套接口的接口建立分布式系统的偏爱。在本章以及下一章中还会出现几个套接口编程的例子，这些例子对你的编程工作非常有帮助。既然本书中所有的例子都是开放源代码软件，所以只要认为对你有帮助，就可以随意重复使用这些程序中的代码。

在本章里，我们将公开两个有用的程序例子：

- 一个对等的客户机/服务器（`client.c` 和 `server.c`）
- 一个简单的可扩展的支持 XML（扩展标记语言）的 Web 服务器（`web_client.c` 和 `web_server.c`）

19.1 套接口的定义

从历史上来讲，TCP 和套接口编程起始于早期的 UNIX 系统。你可以参看有关 UNIX 域套接口（Domain socket）和伯克利套接口（Berkely socket）的参考资料。UNIX 域套接口是为 UNIX 程序间非连网的进程间通信而开发出来的。它们经常用来在 UNIX 系统上实现管道。更现代的伯克利套接口构成了支持现代的 UNIX 系统、Windows、OS/2、Macintosh 和许多其他计算机系统对连网套接口的基础。

当服务器和应用程序需要和其他进程通信时就会创建套接口。从本质上看，一个套接口是进程间通信的端点。每个套接口的名字都是唯一的，所以其他进程能够找到、连接上套接口并且访问它。一对连接的套接口构成了进程间交流数据的一条通信通道，这些进程可以是完全无关的，也能改变数据。服务器程序创建的套接口起到了客户机汇集点的作用。套接口和字符设备有许多共同的特性。但是，套接口只在某个进程与其绑定时才存在。

19.2 通 信 域

通信域用来说明套接口通信协议的语义。每个域都指定了一套协议、控制和解释名字的规则，以及套接口地址的格式。我们将在本章讨论两种类型的域：Internet 域和 UNIX 域。

对于 Internet 域来说，套接口地址的格式是一个 IP 地址和一个端口号。Internet 域套接口，正如其名称所表明的那样，用于连网通信。这些套接口是最常用的套接口类型。它们可以用在几乎任何支持因特网通信的程序上。

另一方面，UNIX 域套接口用于本地进程间通信。它们使用本地路径名作为它们的套接口地址格式。在本章中我们将集中讨论 Internet 域套接口，因为它们是最常用的套接口类型。但是，我们还会介绍一些 UNIX 域套接口的例子作为本章的结束。

19.3 套接口编程基础

在一个程序中使用套接口需要执行 4 个步骤：

1. 分配空间和初始化
2. 连接
3. 传送数据
4. 关闭

在下面的各节里，我们将详细介绍这 4 个步骤。对于每个步骤，我们都会介绍需要用到的系统调用。正如你将要看到的那样，有些步骤随你编写的是客户端程序还是服务器应用而有所不同。

你将会用到系统调用 `socket`、`bind`、`listen`、`connect`、`accept`、`recv` 和 `send` 来编写本章中介绍的几个示例程序。随后的各节在开始介绍示范程序之前，都会简要介绍一下这些系统调用的参数。

19.3.1 分配套接口和初始化

你需要做的第一项工作是分配套接口。接着你就得到了一个套接口描述符。套接口描述符可以比作文件描述符。只要你有套接口描述符，就需要把套接口和预定义的名字空间中的一个名字关联起来。

对于服务器和客户端应用来说，第一步都是一样的。

`socket`

每一个套接口都是一个数据通信通道。在两个进程通过套接口建立连接后，它们就使用套接口描述符来从套接口中读取数据，并向套接口中写入数据。系统调用 `socket` 有以下参数：

```
int domain
int type
```



```
int protocol
```

对套接口来说,有几个可能的域类型。这些类型是在/usr/include/sys/socket.h中定义的。你可以直接在/usr/include/sys/socket.h中找到这些定义,或者在较新版本的C库函数中,可以在OS特定的/usr/include/bits/socket.h中找到,这个文件被包含于/usr/include/sys/socket.h文件中。

- AF_UNIX——UNIX 内部协议
- AF_INET——ARPA 网际协议 (最经常使用的选项)
- AF_ISO——国际标准组织协议
- AF_NS——Xerox 网络系统协议

你可能会常用到 AF_INET 协议。以下是几种类型的套接口:

- SOCK_STREAM——提供了一个可靠的顺序的双向连接 (最常使用的选项)
- SOCK_DGRAM——无连接不可靠的连接
- SOCK_RAW——用于内部网络协议 (root 用户专用)
- SOCK_SEQPACKET——只用于 AF_NS 协议中
- SOCK_RDM——没有实现

你几乎总是会使用 SOCK_STREAM 类型的套接口,因为这个套接口能够在通过套接口连接的进程之间进行双向通信。有些场合可能需要这个参数的其他设置,但是这种情况非常少见。

socket 函数的第三个也就是最后一个参数是协议号。某些套接口类型能够让你在一种以上的协议中进行选择,但是这种情况非常少见。几乎所有的套接口类型都只有一种协议。在所有的示例程序中,这个参数都取 0 值。

bind

函数 bind 将一个进程和一个套接口联系起来。函数 bind 通常用于服务器进程中为接入的客户连接建立一个套接口。函数 bind 的参数如下:

```
int socket
struct sockaddr * my_addr
int my_addr_length
```

函数 bind 的第一个参数是前一个对函数 socket 的调用中返回的套接口值。第二个参数是结构 sockaddr 的地址,结构 sockaddr 是在/usr/include/linux/socket.h中如下定义的:

```
struct sockaddr {
    unsigned short sa_family; // address family, AF_XXX
    char sa_data[14];         // 14 bytes of protocol address
};
```

结构 sockaddr 必须被分配并作为第二个参数传递给函数 bind,但除了初始化数据之外,在示例程序中不能直接访问。例如在例子 server.c 中,我们对结构 sockaddr 进行了如下初始化:

```
struct sockaddr_in sin;  
bzero(&sin, sizeof(sin));  
sin.sin_family = AF_INET;  
sin.sin_addr.s_addr = INADDR_ANY;  
sin.sin_port = htons(port);  
bind(sock_descriptor, (struct sockaddr *)&sin, sizeof(sin));
```

除了为设置协议地址对单独的数据子元素进行命名外,结构 `sockaddr_in` 和结构 `sockaddr` 是等价的。

19.3.2 完成连接的系统调用

如果你正在编写一个客户端应用,下一步是和想与之通信的服务器建立连接。另一方面,如果正在编写一个服务器端应用,就要建立自己的套接口等待来自客户端应用的连接。

头两个系统调用 `listen` 和 `accept` 都用于服务器编程。对于客户端应用来说,只需使用一个用来连接的系统调用。这个调用有个恰当的名字: `connect`。

`listen`

当创建了套接口并且使用 `bind` 把它和一个进程关联起来以后,服务器类型的进程可以调用 `listen` 函数来监听接入的套接口连接。系统调用 `listen` 的参数如下:

```
int socket  
int input_queue_size
```

系统调用 `listen` 的第一个参数是前一次调用 `socket` 函数(而且在调用了 `bind` 之后)返回的整型 `socket` 值。第二个参数设置接入队列的大小。服务器进程经常使用系统调用 `fork` 创建自身的一个副本处理接入的套接口调用;如果你期望同时处理许多客户连接,使用这种方法是相当有效的。然而对于大多数程序来说,通常一次处理一个接入的连接并且把接入队列的大小设置到一个相当大的值就足够了,而且也更简单。

`accept`

当一个接入信号抵达监听套接口,它们会被排入队列直到服务器程序准备好处理它们为止。当服务器准备处理一个新连接时,它会使用系统调用 `accept` 从套接口的队列中检索一个挂起的信号。`accept` 调用会返回一个新的套接口描述符。这个描述符用来进行客户端和服务端应用的通信。同时,原来的套接口仍旧能够监听新的接入信号。

系统调用 `accept` 的参数为

```
int socket  
struct sockaddr *my_addr  
int *my_addr_length
```

`accept` 的第一个参数是监听套接口的套接口描述符。第二个参数是一个指向数据区的指针,它将会把有关接入连接的信息填入到数据区中。第三个参数是一个整数指针,这个整数设置了 `my_addr` 所能容纳的最大字节数。如果 `accept` 调用填入 `my_addr` 的数据量比最大值小,则 `my_addr_length` 的值会变为实际的数据字节数。

某些服务器端程序不使用 `accept` 调用处理接入请求。

`connect`

系统调用 `connect` 用来把本地套接口与远程服务联系起来。正如你将会在本章后面的套接口客户端示例程序中看到的那样，该调用典型的用法是为运行在远程计算机上的一个服务器进程指定本地主机的信息。`connect` 的参数为

```
int socket
struct sockaddr * server_address
int server_address_length
```

`socket` 参数由系统函数 `socket` 的返回值来定义。数据结构 `sockaddr` 将在本章稍后的套接口客户端示例程序中详细讨论。结构 `sockaddr`（族）的第一个数据域允许在调用 `connect` 之前指定连接的类型或者族（没有列出所有的族；参考 `man connect` 窗口完整的列表）：

- `AF_UNIX`——UNIX 域套接口（对于运行在同一台计算机上的进程间高性能的套接口很有用）
- `AF_INET`——Internet IP 协议（这是最常用的族，因为它允许使用一般的 Internet IP 地址进行通信）
- `AF_IPX`——Novell IPX（经常在 Windows 的网络环境中使用）
- `AF_APPLETALK`——AppleTalk 协议

数据结构 `sockaddr` 的第二个数据元素用来指定协议地址。请参考 `bind` 函数的描述，了解有关 `sockaddr_in` 结构的更多信息。在本章 19.4.1 “服务器的例子程序”一节中，读者会看到怎样为 `AF_INET` 族建立协议地址。

19.3.3 传送数据

一旦建立了连接，就开始在服务器和客户机之间传输数据了。有两个系统调用用于传输数据。`recv` 函数用来接收数据。要发送数据，可以使用 `send` 函数。

`recv`

函数 `recv` 用来接收从已经连接的套接口传来的消息，这个套接口已经通过调用 `connect` 和另一个套接口连接起来了。在本例中没有使用另外两个调用 `recvfrom` 和 `recvmsg`，这两个调用是用来从非面向连接的套接口接收消息的。`recvfrom` 将在第 20 章中使用。

`recv` 的参数为

```
int socket
void * buf
int buf_len
unsigned int flags
```

第一个参数定义的套接口必须是已经使用 `connect` 连接到一个端口的套接口。第二个参数是指向内存块的指针，此内存块用来存储接收的信息。第三个参数指定所保留的内存块的大小（以 8 比特的字节为单位）。第四个参数指出了操作标志；下面的值可以结合使用布

尔值和逻辑“与”操作符（`|`）操作作为第四个参数（本章的例子中这个标志均使用零值）。

- `MSG_OOB`——处理带外数据（对于处理高权限控制信息很有用）通常使用 0，执行一般操作（非带外数据）
- `MSG_PEEK`——查看接入消息但不读取它
- `MSG_WAITALL`——返回前等待接收数据的缓冲区被完全填满

send

系统调用 `send` 用来通过套接口向其他程序传递数据。客户端和服务端都使用函数 `send`；客户端应用程序使用 `send` 向远程服务进程传送服务请求，服务器端应用程序使用 `send` 向客户端返回数据。在例子程序中你将看到使用 `send` 的许多例子。函数 `send` 带有以下参数：

```
int socket
const void * message_data
int message_data_length
unsigned int flags
```

第一个参数仅仅是调用函数 `socket` 时返回的套接口值。第二个参数包含了要传送的数据。第三个参数以 8 比特字节为单位指定了信息数据的大小。第四个参数在本章的程序中一直为零，但可以为以下常数（尽管很少使用）：

- `MSG_OOB`——处理带外数据（对于处理高权限控制信息带外 `send` 很有用）通常使用 0 进行一般操作（非带外数据）
- `MSG_DONTROUTE`——不使用路由

19.3.4 关闭

最后，当你用完套接口以后，就该释放你所掌握的套接口了。这可以通过关闭套接口描述符来做到这一点。

close

对这个系统调用没有太多要说的。你可以提供给它一个套接口描述符，而它就会关闭这个套接口。这样做将防止对套接口任何更多的读写操作。如果某个应用试图读写这个套接口，它就会收到一个出错消息。

`close` 惟一的参数是 `int socket`。`socket` 是要关闭的套接口的描述符。

19.4 使用套接口的客户机/服务器例子程序

服务器的示例程序监听一个套接口（端口 8000）等待接入请求。像示例程序 `client.c` 一样，任何程序都可以和这个服务器连接并且上传 16 384 字节的数据。服务器把数据当作 ASCII 数据看待，将其转换为大写后返回给客户端程序。当编写基于套接口的客户机/服务器程序时，能够很容易地重用这两个简单的程序。服务器示例程序没有使用 `fork`（使用 `man`

fork 命令查看它的文档) 来创建它自身的新副本; 它只是建立了一个能够容纳 20 个服务请求的输入队列。那些可能会同时接收许多请求的服务器应该调用 fork 函数创建单独的进程处理计算开销大的服务请求。在某些情况下, fork 调用可能开销太大, 使用线程是更好的想法。要了解线程编程的更多信息, 参见第 14 章。

19.4.1 服务器的例子程序

server.c 创建了一个永久的套接口监听服务请求; 当客户机连接到服务器时, 就建立了一个临时套接口。每次客户机连接到服务器, 一个临时套接口就在客户机和服务器之间打开。接下来的数据既支持为客户连接创建的永久套接口, 又支持临时套接口:

```
struct sockaddr_in sin;  
struct sockaddr_in pin;  
int sock_descriptor;  
int temp_sock_descriptor;  
int address_size;
```

我们必须首先定义套接口描述符:

```
sock_descriptor = socket(AF_INET, SOCK_STREAM, 0);
```

接下来要做的是填写结构 sockaddr_in 必要的域 sin:

```
bzero(&sin, sizeof(sin));  
sin.sin_family = AF_INET;  
sin.sin_addr.s_addr = INADDR_ANY;  
sin.sin_port = htons(8000); // we will use port 8000
```

现在我们可以使用函数 bind 将套接口和端口 8000 连在一起:

```
bind(sock_descriptor, (struct sockaddr *)&sin,  
     sizeof(sin));
```

最后我们开始监听连接到 8000 端口的新套接口:

```
listen(sock_descriptor, 20); // queue up to 20 connections
```

在此处, 例子程序 server.c 进入一个无限的循环来等待来自客户机的套接口接入:

```
while(1) {  
    // get a temporary socket to handle client request:  
    temp_sock_descriptor =  
        accept(sock_descriptor, (struct sockaddr *)  
              &pin, &address_size);  
    // receive data from client:  
    recv(temp_sock_descriptor, buf, 16384, 0);  
    // ... here we can process the client request ...  
  
    // return data to the client:  
    send(temp_sock_descriptor, buf, len, 0);  
    // close the temporary socket (we are done with it)
```

```
        close(temp_sock_descriptor);  
    }
```

现在我们已经解释了程序中所有有意思的部分，现在该是把它们放在一起的时候了。最后完整版本的 server.c 如程序清单 19.1 所示。

程序清单 19.1 程序 server.c

```
/* server.c  
 * Copyright Mark Watson 1999. Open Source Software License.  
 */  
  
#include <stdio.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <arpa/inet.h>  
#include <netdb.h>  
  
int port = 8000;  
  
void main() {  
    struct sockaddr_in sin;  
    struct sockaddr_in pin;  
    int sock_descriptor;  
    int temp_sock_descriptor;  
    int address_size;  
    char buf[16384];  
    int i, len;  
  
    sock_descriptor = socket(AF_INET, SOCK_STREAM, 0);  
    if (sock_descriptor == -1) {  
        perror("call to socket");  
        exit(1);  
    }  
  
    bzero(&sin, sizeof(sin));  
    sin.sin_family = AF_INET;  
    sin.sin_addr.s_addr = INADDR_ANY;  
    sin.sin_port = htons(port);  
  
    if (bind(sock_descriptor, (struct sockaddr *)&sin, sizeof(sin))  
        == -1) {  
        perror("call to bind");  
        exit(1);  
    }  
  
    if (listen(sock_descriptor, 20) == -1) {  
        perror("call to listen");  
        exit(1);  
    }  
  
    print("Accepting connections ...\n");
```

```
while(1) {
    temp_sock_descriptor =
        accept(sock_descriptor, (struct sockaddr *)&pin,
            &address_size);
    if (temp_sock_descriptor == -1) {
        perror("call to accept");
        exit(1);
    }

    if (recv(temp_sock_descriptor, buf, 16384, 0) == -1) {
        perror("call to recv");
        exit(1);
    }
    printf("received from client:%s\n",buf);

    // for this server example, we just convert the
    // characters to upper case:

    len = strlen(buf);
    for (i=0; i<len; i++) buf[i] = toupper(buf[i]);

    if (send(temp_sock_descriptor, buf, len, 0) == -1) {
        perror("call to send");
        exit(1);
    }

    close(temp_sock_descriptor);
}
}
```

在示例程序 `server.c` 中服务器一直在监听等待来自客户套接口的接入。如果程序被强行终止，用于最初与客户端连接的永久套接口将被操作系统自动关闭。在 Linux 下自动关闭操作执行得很快，但在 Windows NT 下就要花费 5 或 10 秒的时间。

19.4.2 客户机的例子程序

为了监听向 `server.c` 中定义的例子发送服务请求，`client.c` 建立了一个临时套接口。以下的数据是用来建立临时套接口与服务器的连接：

```
int socket_descriptor;
struct sockaddr_in pin;
struct hostent *server_host_name;
```

客户机程序必须知道主机的 IP 地址，一般来说这就像 `www.a_company.com` 或在局域网上的名字，诸如 `colossus` 和 `carol`（这是我家的局域网上的我和我妻子的计算机名）。例子程序可以在你的 Linux 计算机上运行，只要你指定本地计算机的标准 IP 地址，例如 `127.0.0.1`，此地址通常是本地主机的别名。在例子 `client.c` 中试图将 `127.0.0.1` 替换为本地主机。以下语句获得主机信息：

```
server_host_name = gethostbyname("127.0.0.1");
```

既然我们已经有了主机信息，我们就可以对结构 `sockaddr_in` 中的数据 `pin` 赋值：

```
bzero(&pin, sizeof(pin));
pin.sin_family = AF_INET;
pin.sin_addr.s_addr = htonl(INADDR_ANY);
pin.sin_addr.s_addr =
    ((struct in_addr *) (server_host_name>h_addr))>s_addr;
pin.sin_port = htons(port);
```

建立与主机进行套接口连接的工作已经准备就绪：

```
socket_descriptor = socket(AF_INET, SOCK_STREAM, 0);
```

最后让我们使用该套接口连接到主机：

```
connect(socket_descriptor, (void *)&pin, sizeof(pin));
```

如果服务器忙，本次调用可能会阻塞（等待）。当对 `connect` 的调用返回时，我们就可以向服务器发送数据了：

```
send(socket_descriptor, "test data", strlen("test data") + 1, 0);
```

接下来对 `recv` 的调用等待服务器的响应（变量 `buf` 定义了长为 8192 字节的数组）：

```
recv(socket_descriptor, buf, 8192, 0);
```

变量 `buf` 中的数据包含了从服务器返回的数据。现在客户机可以关闭与服务器的临时套接口连接了：

```
close(socket_descriptor);
```

对客户机程序的介绍就到这里。在程序清单 19.2 中能够找到该程序的完整代码。

程序清单 19.2 程序 `client.c`

```
/* Client.c
 * Copyright Mark Watson 1999. Open Source Software License.
 * Note: derived from NLPserver client example.
 * See www.markwatson.com/opensource/opensource.htm for all
 * of the natural language server (NLPserver) source code.
 */

#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

char * host_name = "127.0.0.1"; // local host
int port = 8000;

void main(int argc, char *argv[]) {
    char buf[8192];
    char message[256];
```



```

int socket_descriptor;
struct sockaddr_in pin;
struct hostent *server_host_name;

char * str = "A default test string";

if (argc < 2) {
    printf("Usage: 'test \"Any test string\"'\n");
    printf("We will send a default test string.\n");
} else {
    str = argv[1];
}

if ((server_host_name = gethostbyname(host_name)) == 0) {
    perror("Error resolving local host\n");
    exit(1);
}

bzero(&pin, sizeof(pin));
pin.sin_family = AF_INET;
pin.sin_addr.s_addr = htonl(INADDR_ANY);
pin.sin_addr.s_addr = ((struct in_addr *) (server_host_name->
                                           h_addr))->s_addr;
pin.sin_port = htons(port);

if ((socket_descriptor = socket(AF_INET, SOCK_STREAM, 0))
    == -1) {
    perror("Error opening socket\n");
    exit(1);
}

if (connect(socket_descriptor, (void *)&pin, sizeof(pin)) == -1)
{
    perror("Error connecting to socket\n");
    exit(1);
}

printf("Sending message %s to server...\n", str);

if (send(socket_descriptor, str, strlen(str), 0) == -1) {
    perror("Error in send\n");
    exit(1);
}

printf("..sent message.. wait for response...\n");

if (recv(socket_descriptor, buf, 8192, 0) == -1) {
    perror("Error in receiving response from server\n");
    exit(1);
}

printf("\nResponse from server:\n\n%s\n", buf);

close(socket_descriptor);
}

```

19.4.3 运行客户机和服务器的例子程序

如果你是运行 X Windows 系统, 读者可以打开两个 xterm 窗口, 一个用来运行客户机程序。另一个用来运行服务器程序。将目录转换到包含 client.c 和 server.c 的目录 (不必关心此目录的其他文件), 然后键入 make 来编译示例程序。服务器可以通过键入 server 来启动。若要运行客户程序, 在另一个 xterm 窗口中将目录切换正确, 键入:

```
./client "This is test data to send to the server."
```

在 server 运行的 xterm 窗口中, 按下 Control+C 键, 就可以使 server 程序停下来。以下是 server 示例程序的输出:

```
# ./server
Accepting connections ...
received from client:This is test data to send to the server.
```

而在 client 示例程序的窗口中将会有以下输出:

```
# ./client "This is test data to send to the server."
Sending message This is test data to send to the server. to server...
...sent message.. wait for response...

Response from server:

THIS IS TEST DATA TO SEND TO THE SERVER.
```

19.4.4 使用 Web 浏览器作为客户机运行服务器的例子程序

除了使用示例程序 client.c 测试 server.c 外, 我们还可以使用 Web 浏览器。试着打开 Netscape Web 浏览器输入以下地址:

```
http://127.0.0.1:8000
```

在此处, 127.0.0.1 是本地计算机的 IP 地址, 8000 是将要发送到的端口号。Netscape 浏览器将会向 server.c 程序发送请求, 将其假想为一个 Web 服务器。server.c 程序将会接受此请求, 把所有字符转化为大写, 然后返回 Web 浏览器。浏览器上将会显示如下信息:

```
GET / HTTP/1.0 CONNECTION: KEEP-ALIVE USER-AGENT: MOZILLA/4.5 [EN]
(X11; I; LINUX 2.0.35 I686)
HOST:127.0.0.1:8000 ACCEPT: IMAGE/GIF, IMAGE/X-EBITMAP, IMAGE
/JPEG, IMAGE/PJPEG, IMAGE /PNG, /* ACCEPT-ENCODING: GZIP
ACCEPT-LANGUAGE: EN ACCEPT-CHARSET: ISO-8859-1,*,UTF-8
```

很有趣, 是吗?

19.5 一个简单的 Web 服务器和 Web 客户机的例子程序

本节读者会实现一个简单的 Web 服务器和一个基于文本的客户端 Web 应用程序。你既可以使用客户端程序测试 Web 服务程序, 也可以使用 Netscape Navigator 来测试它。这个

例子使用的源代码文件是 web_server.c 和 web_client.c。

19.5.1 实现一个简单的 Web 服务器

示例程序 web_server.c 使用函数 read_file 来读取本地文件，然后把文件的内容当作一个大的字符缓冲区（变量 ret_buf）返回。变量 error_return 存储了 HTML 格式的错误信息值。

以下代码实现了使用函数 read_file 把本地文件的内容读入缓冲区：

```
char ret_buf[32768];
char * error_return = "<HTML>\n<BODY>File not found \n
                      </BODY>\n</HTML>";

char * read_file(char * buf, int num_buf) {
    int i;
    char *cp, *cp2;
    FILE *f;
    cp = buf + 5;
    cp2 = strstr(cp, "HTTP");
    if (cp2 != NULL) *cp2 = '\0';
    if (DEBUG) printf("file:|%s|\n", cp);
    // fetch file:
    f = fopen(cp, "r");
    if (f == NULL) return error_return;
    i = fread(ret_buf, 1, 32768, f);
    if (DEBUG) printf("%d bytes read from file %s\n", i, cp);
    if (i == 0) { fclose(f); return error_return; }
    ret_buf[i] = '\0';
    fclose(f);
    return ret_buf;
}
```

在例子 web_server.c 中函数 main 打开一个套接口，以通常的方式监听服务请求。web_server 检查所有系统调用的错误代码；在下面对 Web 服务器如何工作的简短描述中，这些检查均被忽略。以下的代码和示例程序 server.c 中服务器套接口的建立代码很相似：

```
int sock;
int serverSocket;
struct sockaddr_in serverAddr;
struct sockaddr_in clientAddr;
int clientAddrSize;
struct hostent* entity;
serverSocket = socket(AF_INET, SOCK_STREAM, 0);
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(port);
serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
memset(&(serverAddr.sin_zero), 0, 8);
```

接下来对 `bind` 的调用把套接口和希望连接的端口号（来自变量 `port`）连接起来：

```
bind(serverSocket, (struct sockaddr*) &serverAddr,
    sizeof(struct sockaddr));
```

然后是调用 `listen`，本调用建立了一个上限为 10 的请求队列，这表明程序已经为接受服务请求做好了准备：

```
listen(serverSocket, 10); // allow 10 queued requests
```

现在例子 `web_server.c` 开始循环，等待接入连接。对每一个接入的服务请求都有一个新的套接口被打开；服务器读取服务请求的数据，然后将数据返回客户机作为对请求的处理，最后临时套接口（变量 `sock`）被关闭。

接下来的 `web_server.c` 中的代码片段实现了一个处理循环来等待接入客户请求：

```
while (1) {
    clientAddrSize = sizeof(struct sockaddr_in);
    do
        sock = accept(serverSocket,
            (struct sockaddr*) &clientAddr,
            &clientAddrSize);
    while ((sock == -1) && (errno == EINTR));
    if (sock == -1) {
        printf("Bad accept \n");
        exit(1);
    }

    entity = gethostbyaddr((char*) &clientAddr.sin_addr,
        sizeof(struct in_addr), AF_INET);
    if (DEBUG) printf("Connection from %d \n", inet_ntoa,
        ((struct in_addr) clientAddr.sin_addr));

    i = recv(sock, recvBuffer, 4000, 0);

    if (i == -1) break;
    if (recvBuffer[i - 1] != '\n') break;
    recvBuffer[i] = '\0';
    if (DEBUG) {
        printf("Received from client: %s\n", recvBuffer);
    }

    // call a separate work function to process request:
    cbuf = read_file(recvBuffer, totalReceived);
    size = strlen(cbuf);
    totalSent = 0;
    do {
        bytesSent = send(sock, cbuf + totalSent,
            strlen(cbuf + totalSent), 0);
        if (bytesSent == -1) break;
        totalSent += bytesSent;
    }
```

```

    } while (totalSent < size);
    if (DEBUG) printf("Connection closed by client.\n");
    close(sock);
}

```

这段代码和例子程序 `server.c` 非常类似, 区别在于从客户端机接收的数据被解释为对本地文件的请求。被请求的本地文件被读入一个缓冲区, 然后将缓冲区的数据发送回客户机。

该是做总结的时候了。程序清单 19.3 给出了完整的程序。注意, 这个程序虽然短但是功能非常强大。

程序清单 19.3 程序 `web_server.c`

```

/* web_server.c
 * Copyright Mark Watson 1999. Open Source software license.
 */
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>

unsigned short port = 8000;    // Default server port number

#define DEBUG 1
char * error_return = "<HTML>\n<BODY>File not found\n</BODY>\n\n</HTML>";

char ret_buf[32768];

char * read_file(char * buf, int num_buf) {
    int i;
    char *cp, *cp2;
    FILE *f;
    cp = buf + 5;
    cp2 = strstr(cp, " HTTP");
    if (cp2 != NULL) *cp2 = '\0';
    if (DEBUG) printf("file: |%s|\n", cp);
    // fetch file;
    f = fopen(cp, "r");
    if (f == NULL) return error_return;
    i = fread(ret_buf, 1, 32768, f);
    if (DEBUG) printf ("%d bytes read from file %s\n", i, cp);
    if (i == 0) { fclose(f); return error_return; }
    ret_buf[i] = '\0';
    fclose(f);
    return ret_buf;
}

int main(int argc, char* argv[]) {

```

```
int i, sock;
char* recvBuffer = (char *)malloc(4001);

int rc = 0;
int serverSocket;
struct sockaddr_in serverAddr;
struct sockaddr_in clientAddr;
int clientAddrSize;
struct hostent* entity;
int totalReceived;
int size;
int totalSent;
int bytesSent;
char * cbuf;

serverSocket = socket(AF_INET, SOCK_STREAM, 0);
if (serverSocket == -1) {
    printf("Invalid socket\n");
    exit(1);
}

serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(port);
serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
memset(&(serverAddr.sin_zero), 0, 8);

printf ("Binding server socket to port %d\n", port);

rc = bind(serverSocket, (struct sockaddr*) &serverAddr,
          sizeof(struct sockaddr));
if (rc == -1) {
    printf("Bad bind\n");
    exit(1);
}

rc = listen(serverSocket, 10); // allow 10 queued requests
if (rc == -1) {
    printf("Bad listen\n");
    exit(1);
}

printf("Accepting connections ...\n");

while (1) {
    clientAddrSize = sizeof(struct sockaddr_in);
    do
        sock = accept(serverSocket,
                      (struct sockaddr*) &clientAddr,
                      &clientAddrSize);
    while ((sock == -1) && (errno == EINTR));
    if (sock == -1) {
```

```

        printf("Bad accept\n");
        exit(1);
    }

    entity = gethostbyaddr((char*) &clientAddr.sin_addr,
        sizeof(struct in_addr), AF_INET);
    if (DEBUG) printf("Connection from %d\n",
        inet_ntoa((struct in_addr) clientAddr.sin_addr));

    i = recv(sock, recvBuffer, 4000, 0);
    if (i == -1) break;
    if (recvBuffer[i - 1] != '\n') break;
    recvBuffer[i] = '\0';
    if (DEBUG) {
        printf("Received from client: %s\n", recvBuffer);
    }

    // call a separate work function to process request:
    cbuf = read_file(recvBuffer, totalReceived);
    size = strlen(cbuf);
    totalSent = 0;
    do {
        bytesSent = send(sock, cbuf + totalSent,
            strlen(cbuf + totalSent), 0);
        if (bytesSent == -1) break;
        totalSent += bytesSent;
    } while (totalSent < size);
    if (DEBUG) printf("Connection closed by client.\n");
    close(sock);
}

return 0;
}

```

19.5.2 实现一个简单的 Web 客户机

简短示例程序 `web_client.c` 演示了怎样才能在程序中和一个 Web 服务器相互作用，而不是使用 Web 浏览器。示例程序检测了所有来自系统调用的错误返回值，但是为使得程序清单简短易读，在下面的讨论中这些错误检查都被忽略。通常返回小于零的错误标志都说明存在问题。

变量 `host_name` 和 `port` 用来指定作为主机的计算机的名字和端口。在例子 `web_server.c` 中主机名字被指定为本地计算机的绝对的 IP 地址 (127.0.0.1)，当然，将其指定为其他的值，诸如 `www.lycos.com` 或 `www.markwaston.com` 也同样工作。默认情况下，Web 服务器监听 80 端口。端口 8000 用于运行 `web_server.c`，以免和大部分 Linux 发布中默认方式下安装和运行的 Apache Web 服务器冲突：

```

char * host_name = "127.0.0.1"; // local host
int port = 8000;

```

下面的代码看上去和我们介绍过的其他例子很相似。参见程序清单 19.4 了解完整的代码。

程序清单 19.4 程序 web_client.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

char * host_name = "127.0.0.1"; // local host
int port = 8000;

main(int argc, char *argv[]) {
    char buf[8192];
    char message[256];
    int sd;
    struct sockaddr_in pin;
    struct hostent *nlp_host;

    if ((nlp_host = gethostbyname(host_name)) == 0) {
        printf("Error resolving local host\n");
        exit(1);
    }

    bzero(&pin, sizeof(pin));
    pin.sin_family = AF_INET;
    pin.sin_addr.s_addr = htonl(INADDR_ANY);
    pin.sin_addr.s_addr = ((struct in_addr *) (nlp_host->h_addr))
        ->s_addr;
    pin.sin_port = htons(port);

    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        printf("Error opening socket\n");
        exit(1);
    }

    if (connect(sd, (void *)&pin, sizeof(pin)) == -1) {
        printf("Error connecting to socket\n");
        exit(1);
    }

    // NOTE: must send a carriage return at end of message:
    sprintf(message, "GET /index.html HTTP/1.1\n");
    printf("Sending message %s to web_server...\n", message);
    if (send(sd, message, strlen(message), 0) == -1) {
        printf("Error in send\n");
        exit(1);
    }
}
```



```

    }

    printf("..sent message.. wait for response...\n");

    if (recv(sd, buf, 8192, 0) == -1) {
        printf("Error in receiving response from NLPserver\n");
        exit(1);
    }

    printf("\nResponse from NLPserver:\n\n%s\n", buf);

    close(sd);
}

```

19.5.3 测试 Web 服务器和 Web 客户机

打开两个 xterm 窗口，将目录均转换到 IPC/SOCKETS 目录下，在第一个窗口中键入：

```

make
./web_server

```

在第二个窗口中键入：

```
./web_client
```

你可以在示例程序 web_server.c 运行的窗口中看到以下的输出：

```

# ./web_server
Binding server socket to port 8000
Accepting connections ...
Connection from 1074380876
Received from client: GET /index.html HTTP/1.1

file: |index.html|
473 bytes read from file index.html
Connection closed by client.

```

而在示例程序 web_client 运行的窗口中你可以看到：

```

# ./web_client
Sending message GET /index.html HTTP/1.1
to web_server...
..sent message.. wait for response...
Response from NLPserver:
<HTML>
  <HEAD>
    <TITLE>Mark's test page for the web_server</TITLE>
  </HEAD>
  <BODY>
    <H2>This is a test page for the web_server</H2>
    <p>In order to test with another local file, please
    <a href="test.html">click here</a> to load a local

```

```
file (local to web_server).<p>
In order to test the web server with a remote link,
here is a link to Mark Watson's home page on the net.<p>
Please <a href="http://www.markwatson.com">client here</a>.
</BODY>
</HTML>
```

19.5.4 使用 Netscape Navigator 作为客户机运行简单的 Web 服务器

你可以用 Web 浏览器如 Netscape Navigator 来测试这里的 Web 服务器示例。打开 Web 浏览器然后输入 URL, `http://127.0.0.1:8000/index.html`。

注意, 你必须在 URL 中指定一个文件名, 因为 Web 服务器示例不会尝试去打开像 `index.html` 或 `index.htm` 这样的默认值。

19.6 通过其他编程语言使用套接口

即使许多人选择用 C 语言完成他们所有的编程工作, 但是在 UNIX 平台上产生的大多数语言都提供了对套接口编程很好的支持。这是因为套接口是 UNIX 操作系统本身的一部分。C++ 对套接口的支持和 C 的支持一样好, 因为你可以在 C++ 中使用和 C 一样的技术。但是, C++ 还有几个实现套接口编程的类库。这样你就可以用面向对象技术进行套接口编程。Common C++ 就是这些类库中的一个很好的例子, 可以在 `http://sourceforge.net/projects/cplusplus/` 下找到它。在 Troll Tech 的 Qt 窗口部件库中也提供了一些对套接口的支持。参见第 28 章“使用 Qt 进行 GUI 编程”了解有关 Qt 库的更多信息。

Java 对套接口的支持极其出色, 因为它是专门为 Internet 设计的语言。你极有可能发现它对套接口和多线程的支持让它成为开发基于 Internet 的应用的理想语言。

Python 用它的 `socket` 模块提供了对低层次的套接口编程的支持。通过这个模块可以访问 `socket` 对象类。`socket` 类实现的功能和本章我们介绍过的 C 支持的套接口功能类似。Python 也有许多用于套接口编程的较高层次的类, 并且实现了许多基于套接口的协议。使用 Python 语言开发 Internet 应用非常简单。

Perl 是在 UNIX 上的多种脚本和系统维护工作中广泛使用的语言。它提供了良好而可靠的套接口支持。它通过源自 C 的接口对低层次的套接口编程提供支持。和 Python 的情况类似, 它也有一些较高层次的函数和类可以简化套接口编程工作。Perl 发布版本中包含了许多基于套接口的高层协议, 也可以从 CPAN 库中得到它们。在下一节中我们将在所有的例子中使用 Perl。

19.7 UNIX 域套接口的 Perl 编程

正如已经介绍过的那样, UNIX 域套接口只能用于内部通信。它们主要用于进程间通信 (interprocess communication, IPC), 而很少在应用程序中使用。如果你需要在一个应用

程序中使用套接口通信，通常要使用 Internet 域套接口而不是 UNIX 域套接口。这样做的原因有几点。首先，使用 Internet 域套接口的应用程序既能在本地的环境也能在连网环境下工作。通过这种方法，你就能充分利用两种世界。其次，许多套接口编程的支持库都只能支持 Internet 域套接口，或者至少对 Internet 域套接口的支持更好。

也存在 UNIX 域套接口相当有用的场合。UNIX 域套接口的知识迟早也会有用的，特别是如果你要用较老的程序或者低层次的编程来工作，更是如此。在没有连网功能的环境中使用的工具，或者你认为网络协议额外开销太大，就是要使用 UNIX 域套接口场合的例子。

使用两个参数，一个 IP 地址和一个端口号来定义一个 Internet 域套接口。要定义 UNIX 域套接口，需使用一个文件名，如果愿意可以使用路径名构成完整的文件名。一旦套接口构建好了，这个文件也就构建好了而且会出现在文件系统中。因为 UNIX 域套接口是被映像到文件系统的，所以你可以使用 ls 命令来查看它们。例如：

```
# ls -l /tmp/testsocket
srwxrwxr-x  1 patrikj patrikj  0 Sep  8 00:26 /tmp/testsocket
```

文件标志中的 s 告诉你这个文件是一个套接口，而不是普通文件。

如果留意过多种 IPC 机制，你会发现一个 UNIX 域套接口的行为在很多方面都和一个有名管道相同。但是使用命名管道不能把一个进程的数据从另一个进程的数据中区分出来。使用 UNIX 域套接口就能为每个进程获得一个单独的会话。

我们在示例中不使用 C 而使用 Perl 语言。这会暂时中断迄今为止在所有示例中使用 C 代码的传统，这个程序表明使用 Perl 进行套接口编程比较容易。这个例子使用低层次的套接口 API 编写代码，这和你在本章前面的部分中看到的 C 示例程序非常相似。程序中也有 Perl 中面向对象的套接口编程工具。

在示例中，我们将创建一个简单的服务器，它等待 UNIX 域套接口上的请求。一旦有这样的请求抵达，它就向客户机返回一个应答然后切断连接。这个程序的功能并不复杂，但是它能向你展示如何使用 UNIX 域套接口的基础知识。在程序清单 19.5 中可以找到本例的服务器代码。

程序清单 19.5 程序 server.pl

```
#!/usr/bin/perl -w

use Socket;

$socketname="/tmp/request";
unlink($socketname);

$addr = sockaddr_un($socketname);
socket(REQSOCK, PF_UNIX, SOCK_STREAM, 0) || die "No socket:$!";
bind (REQSOCK,$addr) || die "Can't bind:$!";
listen (REQSOCK,5) || die "Can't listen:$!";

for($waiting=0;
    accept(REQUEST,REQSOCK) || $waiting;
    $waiting = 0, close REQUEST) {
```

```
    next if $waiting;

    print REQUEST "Hi, there client.\nBye for now!\n";
}
```

从这个例子里你会注意到它或多或少和前面的例子有些相同。即使你不懂 Perl 语言，我想你阅读上面这个简单的程序也不会有任何问题。除了使用 Perl 语言外，本例中，不引入任何新概念。正如你所看到的那样，UNIX 和 Internet 域套接口在编程方面非常兼容。注意，在我们尝试打开套接口之前，我们需要确保借助 unlink 调用删除任何老版本的套接口。如果我们没有这样做，系统会认定套接口处于使用状态而执行失败。

正如你在程序清单 19.6 中看到的那样，这个例子的客户端部分甚至比服务器部分更简单。它会连接到服务器的套接口上，然后显示通过套接口返回的所有信息，直至套接口被关闭。

程序清单 19.6 程序 client.pl

```
#!/usr/bin/perl -w

use Socket;
socket(REQSOCK, PF_UNIX, SOCK_STREAM, 0)
    || die "No socket: $!";
connect(REQSOCK, sockaddr_un("/tmp/request"))
    || die "Can't connect: $!";

while (<REQSOCK>) {
    print;
}
```

正如你从这个例子所看到的，从 Perl 中使用套接口非常简单。只要简单的几行，你就能在脚本中使用强大的 IPC 机制。

19.8 监视套接口活动的工具

有两种 UNIX 工具可以用来在你的系统上监视套接口的行为。这两个工具是 netstat 和 tcpdump。tcpdump 只能用于检查 Internet 域套接口，因为它只能监视网络流量。当你调试自己的套接口应用时，这两条命令都可以作为好帮手。

netstat 工具能够查看系统上打开的套接口连接。使用带有 -a 选项的 netstat 查看所有的连接：

```
netstat -a
```

netstat 命令有许多能够改变其行为的开关。参数 -A 用来设置套接口所属的域类型。由于 netstat 命令的开关和参数太多，所以本章不能全部介绍它们。可使用 man netstat 了解更多有关这个工具的信息。

要使用 `tcpdump` 命令你必须作为超级用户登录系统，因为网络接口必须被设置为混杂（promiscuous）模式。这个命令能够用来监视通过计算机上网络接口的所有网络流量。注意，这会产生大量的输出。

如果你打算使用 `tcpdump` 监视套接口流量，可能只想显示包含你的应用程序使用的套接口的流量。工具 `tcpdump` 有一种小语言，用这种小语言编写的表达式能够选出你所感兴趣的数据包。要了解更多的相关信息，可以参考 `tcpdump` 的手册页面。

下面的例子显示了我怎样使用 `tcpdump` 截获从浏览器到本地 Web 服务器的所有数据包的。所有数据包都通过管道送入 `hexdump`，这个程序将数据流以一种良好而可读的格式输出。

```
# tcpdump -lw - "src host schismatrix.gnunix.org and \
dst host schismatrix.gnunix.org and dst port 80" |
hexdump -e '"%06.6_a0: " 10/1 "%04x " "\t" 20/1 "%_p" "\n"'

001240: 0073 0070 0063 0068 002e 006a 0073 0020 HTTP/1.0..If-Mod
001270: 0069 0066 0069 0065 0064 002d 0053 0069 nce: Tue, 05 Sep
001320: 0020 0032 0030 0030 0030 0020 0032 0032 :29:59 GMT; leng
001350: 0074 0068 003d 0037 0038 0034 003f 009e .9L.....
001400: 0000 0000 0008 0000 0000 0000 0000 0000 .....E...
001430: 002f 00c5 0040 0000 0040 0006 00da 004e .UCB.UCB...Psl..
001460: 0073 00e8 00a8 0024 0080 0018 0079 0060 .....;^.,;S
001510: 0047 0045 0054 0020 002f 007e 0070 0061 trikj/spch.js HT
001540: 0054 0050 002f 0031 002e 0030 000d 000a If-Modified-Sinc
001570: 0065 003a 0020 0054 0075 0065 002c 0020 05 Sep 2000 22:2
001620: 0039 003a 0035 0039 0020 0047 004d 0054 ; length=784?..9
```

19.9 小 结

在本章里，通过客户端和服务器的示例程序，你学到了如何使用 TCP 套接口编写程序。套接口编程是大多数分布式系统的基础。而其他的高层技术，比如 RPC、CORBA 和 Java 的 RMI 也很流行，但是大多数现有的系统都是使用套接口写成的。即使你使用了一些高层次技术，但是与它们相比套接口的效率更高，所以套接口编程是应该了解的一种好技术。

第 20 章 UDP：用户数据报协议

本章向你展示如何使用 UDP——用户数据报协议。本章以概述开始，对比了 UDP 和 TCP，然后提供了一个示例程序。

20.1 UDP 概述

第 19 章“TCP/IP 和套接口编程”向你展示了如何使用 TCP（传输控制协议）进行通信。UDP（用户数据报协议）提供的服务很少，要求程序员进行更多的控制。因此，它被看作是一种更低层的协议。

通常在 Linux 和 UNIX 中实现的因特网协议由四层组成。最上面一层是应用，或者说是用户代码。应用使用 TCP 或 UDP 协议进行通信。TCP 和 UDP 使用 IP（网际协议）。IP 和网络接口（可能通过设备驱动程序）——物理链接，比如以太网、ATM 或者 FDDI 进行通信。在每一层中，处理的内容越来越细致。应用程序对数据包的大小、路由和重传一无所知。TCP 处理许多这样的细节。UDP 处理的细节要少些，但在正确实现的情况下能以更少的网络负载充分提高速度。

20.1.1 UDP 和 TCP 的对比

UDP 处理的细节比 TCP 少。UDP 不能保证消息被传送到（它也不报告消息没有传送到）目的地。UDP 也不保证数据包的传送顺序。传送顺序很重要，因为 IP 可以把用户数据分成多个物理的数据包，而且可以通过不同的路由传送到目标系统。

TCP 处理 UDP 不处理的细节。TCP 被认为是“面向连接”的协议，而 UDP 被认为是“无连接”的协议。TCP（通过重传以及报告传输失败和连接中断）保持一个连接。UDP 把数据发送出去以后只能希望它能够抵达目的地。

使用 TCP 和打电话类似——有信号可以让你知道电话的另一端有应答，有方法可以判断你是否找对了人，有终止信号，甚至还有出错信号（类似电话忙音）。UDP 和在大厅中喊话类似——你实际上不知道自己的话是否被人听到，也不知道你要联系的人是否确实在。结果，如果细节对应用程序比较重要，则程序员必须自己处理这些细节。

20.1.2 TCP 的优缺点

大多数程序员在编写使用套接口通信的程序时都愿使用 TCP，因为 TCP 处理了许多细节。这也是为什么我们不直接使用 IP 或原始的物理网络设备的原因——因为我们在所有时间里都不需要处理所有细节。

TCP 的优点有：

- TCP 提供以认可的方式显式地创建和终止连接。

- TCP 保证可靠的、顺序的（数据包以发送的顺序接收）以及不会重复的数据传输。
- TCP 处理流控制。
- TCP 允许数据优先。
- 如果数据没有传送到，则 TCP 套接口返回一个出错状态条件。
- TCP 通过保持连接并将数据块分成更小的分片来处理大数据块——无需程序员知道这一情况。

当然，TCP 不是完美的。否则就没人会用 UDP 了。TCP 最大的缺点是在转移数据时必须创建（并保持）一个连接。这个连接给通信进程增加了开销，让它比 UDP 的速度要慢。

20.1.3 UDP 的优缺点

正如比较两种极端选择的时候常见的那样，一种选择的优点往往就是另一种选择的缺点。

UDP 的优点有：

- UDP 不要求保持一个连接。
- UDP 没有因接收方认可收到数据包（或者当数据包没有正确抵达而自动重传）而带来的开销。
- 设计 UDP 的目的是用于短应用和控制消息。
- 在一个数据包接一个数据包的基础上，UDP 要求的网络带宽比 TCP 更小。

UDP 的缺点有：

- 程序员必须创建代码检测传输错误并进行重传（如果应用程序要求这样做）。
- 程序员必须把大数据包分片。

第 19 章已集中讨论了 TCP。本章集中讨论 UDP。

20.1.4 选择使用哪一种协议

你要作出一个选择：使用 TCP 还是 UDP。你该如何决定使用哪一种协议呢？最简单的答案是需要看看你的应用程序是怎样工作的、数据的重要性、数据的传送量以及要求数据完整性的程度。

选择 UDP 的一个充分理由是应用程序发送周期性的状态消息，这些消息重要程度不高或者有规律地重复。定期向其他用户或者中心服务器广播每个用户状态的多用户游戏就是一个良好的候选应用程序。如果一次更新信息丢失，随后很快（在几分之一秒内）就会有一次更新。在这种类型的应用程序中，传送的每个数据项都相当小——不是从 Web 服务器来的向客户提供服务的一个图像文件。

另一方面，如果你要传输一幅图像或者一个重要的数据集，丢失一点数据就会破坏整个传输，那么你需要 TCP 的可靠性。这一点在应用程序本身不能指出丢失数据的时候尤为重要。一个应用程序可以通过 UDP 实现 TCP 的可靠性，但是要由程序员执行错误检测和重传。

你可能会注意到，像 Telnet（终端模拟）和 Web 服务器/浏览器这样的被认为是标准因

特网应用程序的大都基于 TCP。如果你键入的命令没有被正确地发送，或者 HTML 或 Web 页面在传输过程中被弄乱，这的确会令人厌烦。因此，这些工具都使用 TCP。

在多用户游戏中，丢失来自某个用户的状态更新（丢失 UDP 数据包）可能不会引起注意。下一次传输包含了相同的数据域（可能带有不同的值）能让程序继续玩下去而不会让任何人注意。

传输介质和距离也是考虑的因素。设计用在局域网（LAN）内的多用户游戏比基于因特网的游戏更适合使用 UDP。因为在 LAN 中丢失数据包的可能性更低。

由你来决定选择什么协议：TCP 还是 UDP。对于合适的应用来说，UDP 是一种强大的协议，因为它减少了开销。

20.2 实现一个基于 UDP 的应用

你会注意到下面几节中使用 UDP 的代码所调用的函数和用于 TCP 的函数非常类似。这主要因为套接口库在低层的 TCP 和 UDP 函数上加入了一层抽象。通过这层抽象，你获得了编程更容易（速度更快）的好处，但也失去了一些控制能力。

在使用 TCP 和使用 UDP 之间，函数调用的惟一实际区别在于 socket 函数调用的一个参数。这个参数为 SOCK_STREAM 时代表 TCP，而 SOCK_DGRAM 代表 UDP。对于 TCP 和 UDP 你都可以使用 recvfrom 函数，但是 recv 只能用于 TCP。

当然，程序的变化不是那样简单，因为你要从一种有连接的协议转向一种无连接的协议上去。

20.2.1 使用 UDP 发送数据

最简单的 UDP 应用是一个启动向另一个系统的某个端口传输消息的程序。这个程序没有握手机制，也不确认另一个系统是否正在监听、接收或者处理数据。但它是个良好的开端。

程序清单 20.1 所示的程序 sender.c 发送 20 个文本消息然后再发送一个终止消息（让接收方知道它发送完了）。

程序清单 20.1 sender.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

int port = 6789;

void main() {
    int socket_descriptor;
```



```
    int iter = 0;
    char buf[80];
    struct sockaddr_in address;

/*
    Initialize socket address structure for Internet Protocols
*/
    bzero(&address, sizeof(address)); /* empty data structure */
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr("127.0.0.1");
    address.sin_port = htons(port);
/*
    Create a UDP socket
*/
    socket_descriptor = socket(AF_INET, SOCK_DGRAM, 0);
/*
    Loop 20 times (a nice round number) sending data.
*/
    for (iter = 0; iter <= 20; iter++)
    {
        sprintf(buf, "data packet with ID %d\n", iter);
        sendto(socket_descriptor,
                buf, sizeof(buf),
                0, (struct sockaddr *)&address, sizeof(address));
    }
/*
    send a termination message
*/
    sprintf(buf, "stop\n");
    sendto(socket_descriptor,
            buf, sizeof(buf),
            0, (struct sockaddr *)&address, sizeof(address));

    close(socket_descriptor);
    printf("Messages Sent, Terminating\n");
    exit (0);
}
```

程序 `sender.c` 看上去和 TCP 套接口的例子很像, 区别在于, 当调用 `socket` 函数时 UDP 使用 `SOCK_DGRAM` 而不是 `SOCK_STREAM` 作为其第二个参数。它的第一个参数 (地址族) 指定为 `AF_INET`, 所以我们的套接口是在 IP 上工作的, 不只限于在本机上使用 (正如使用 UNIX 域套接口的情况)。

注意: 函数 `inet_addr` 被 Linux 下的 `inet_aton` 所取代。查看手册页面了解更多的信息。目前, 在非 Linux 的操作系统中, 提供 `inet_addr` 的系统比提供 `inet_aton` 的系统更多。

提示： 这些例子都使用 127.0.0.1 作为本机 IP 地址。这个 IP 地址比较特殊，它被用作环回地址——通常指示本机系统。每个系统都把 127.0.0.1 定义为系统本身，所以，不管你怎样称呼自己的 Linux 机器，也不管它的 IP 地址是什么，这些例子应该都能工作。当网络中只有一台机器，你也可以运行这些例子。在实际的应用程序中，主机名可以作为命令行参数传递给程序，也可以从配置文件中读取，或者从 GUI 对话框中获得。为了简单起见，这些例子都在代码中直接使用地址。

如果你在多台主机上运行示例程序，你要改变主机地址和程序匹配。

`inet_addr` 函数把作为参数的字符串（以点分十进制数表示的 IP 地址）转换为内部能够使用的一个作为因特网地址的整数。为了使用实际的主机名（比如 `macmillanusa.com`），需要编写代码查询域名服务器（DNS）以获得类似 209.238.119.186 那样的绝对 IP 地址。

`htons` 函数把作为参数传递给它的整数从主机字节序转换为网络字节序。因为不同的系统对数据的内部表示方法也不同，系统本身有个标准次序。这个函数负责处理这种转换，所以你不需关心细节。因此你就不必担心你的主机字节序是否和网络字节序匹配了，总是会用到这个函数的。如果数据已经处于正确的字节序了，函数调用也不会带来损害。

`sendto` 函数通过套接口发送一条消息。此时使用了无连接的套接口：消息发送的地址由 `address` 指定。

注意： 在 `sender.c` 和 `receiver.c` 中使用的端口必须一致，都是 6789，否则它们就不能进行通信。你可以把端口想像成电视机的频道或收音机的频率。如果发送方和接收方的选择不同，就不会通信。

从 0 ~ 1024 的端口为 UNIX 服务保留。为了绑定到这些端口之一，必须以超级用户身份运行。文件 `/etc/services` 列出了在你的系统上分配的端口。任何时候安装套接口程序（按常规运行的程序，而不是本书的例子）都必须向 `/etc/services` 加入其端口信息。

你的程序不应该使用任何其他应用正在使用的端口。否则，因为有不同的应用，你的发送程序会结束和接收程序之间的通信。

for 循环用 UDP 发送 20 个消息。消息简单地只随文本包含消息的引用编号。当循环完成后，发送的最后一条消息带有单词 `stop`，于是接收方知道何时停止等待更多的数据。单词 `stop` 没有什么特殊之处，但它的特定含义必须为发送方和接收方共同理解。

C 的 `sprintf` 函数用于从格式化的文本（把整数变量 `inter` 转换为等价的 ASCII）构成一个字符串。

要编译这个程序，应该使用

```
cc -o sender sender.c
```

注意： 有些系统（Linux 不在其中）要求你在编译过程中指定 Internet 库。如果你得到了找不到 `socket` 函数的错误信息，就要在命令行中包含 `-lxnet` 选项：

```
cc -lxnet -o sender sender.c
```

注意： 当你编译这些程序时，你会得到类似下面的警告：

```
sender.c: In function 'main':
sender.c:11:warning: return type of 'main' is not 'int'
```

这是因为我习惯于声明 main 的类型为 void 而不是 int，而且使用 exit 函数而不是 return 退出。这是编程风格的事。

要运行这个程序，只需输入 sender 即可。如果 sender 的执行一切顺利，除了告诉你它已经结束的消息之外你不会看到其他任何消息。

20.2.2 接收 UDP 数据

另一个使用 UDP 的最简单的应用是一个监听程序，它在某个端口上监听从其他系统传来的消息。对这个例子来说，程序中没有尝试验证发送系统、用户甚至应用程序的身份。我们希望能是 sender.c 在发送数据。虽然非常简单，但它是个好开端。

程序清单 20.2 所示的程序 receiver.c 布置了一个无穷的 while 循环，等待接收数据。当它得到数据时，就把数据显式出来。在从发送方得到终止消息之前，它一直都这么做。

程序清单 20.2 receiver.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

int port = 6789;

void main() {
    int sin_len;
    char message[256];
    int socket_descriptor;
    struct sockaddr_in sin;
    printf("Waiting for data from sender\n");
    /*
     * Initialize socket address structure for Internet Protocols
     */
    bzero(&sin, sizeof(sin)); /* empty data structure */
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(INADDR_ANY);
    sin.sin_port = htons(port);
    sin_len = sizeof(sin);
    /*
     * Create a UDP socket
     * and bind it to the port
     */
```

```
socket_descriptor = socket(AF_INET, SOCK_DGRAM, 0);
bind(socket_descriptor, (struct sockaddr *)&sin, sizeof(sin));
/*
Loop forever (or until a termination message is received)
Receive data through the socket and process it. The processing
in this program is really simple -- printing.
*/
while(1)
{
    recvfrom(socket_descriptor, message, sizeof(message), 0,
             (struct sockaddr *)&sin, &sin_len);
    printf("Response from server:%s\n", message);
    if (strncmp(message, "stop", 4) == 0)
    {
        printf("Sender has told me to end the connection\n");
        break;
    }
}
close(socket_descriptor);
exit(0);
}
```

程序 receiver.c 看上去和 TCP 套接口的例子很像, 区别在于, 当调用 socket 函数时 UDP 使用 SOCK_DGRAM 而不是 SOCK_STREAM 作为其第二个参数。它的第一个参数 (地址族) 指定为 AF_INET, 所以我们的套接口是在 IP 上工作的, 不只限于在本机上使用 (正如使用 UNIX 域套接口的情况)。

函数 htonl 把作为参数传递给该函数的长整数 (long 类型) 从主机字节序转换为网络字节序。特殊值 INADDR_ANY 用来告诉 bind 函数请求可以来自于任何 IP 地址。你已经看到过用于短整数 (short 类型) 的 htons 函数了。

bind 函数给套接口分配一个地址。

函数 recvfrom 等待来自于无连接套接口的一个数据包。它也在地址数据结构中返回发送方的地址。这个源地址在示例程序中没有用, 但是更高级的程序能够用它打开一个返回发送方的套接口。如果你不关心这个地址, 可以给参数传递 NULL。

while 循环永远运行, 直至接收到终止消息。随着每次接收消息, 它都把接收到的消息显示出来。消息本身非常简单。它们包含带有消息引用编号的文本。终止消息由单词 stop 构成, 它很重要, 因为发送方和接收方对其特殊的含义理解一致。

警告: 如果发送方使用其他文字而不是 stop 作为终止通知消息, 或者终止消息丢失, 你认为会发生什么情况呢? 非常简单, receiver 会运行并等待、等待、再等待, 直到你取消它为止。

另一个描述等待数据的术语为阻塞 I/O。防止 (阻塞) 程序继续执行, 直到接收到数据为止。

参考 20.2.4 节 “非阻塞 I/O” 了解更多信息。

要编译这个程序，应该使用

```
cc -o receiver receiver.c
```

要运行它，只需简单地键入 receiver。

在系统上测试这个程序最简单的办法就是打开两个命令提示窗口，在不同的窗口中运行每个程序。首先要启动 receiver，因为 sender 不会检查在另一端是否有人。在你运行 receiver 的窗口中，你会看到类似下面的显示：

```
Waiting for data from sender
Response from server: data packet with ID 0
Response from server: data packet with ID 1
Response from server: data packet with ID 2
Response from server: data packet with ID 3
Response from server: data packet with ID 4
```

只要发送方发出的数据包抵达接收方，这个输出就会持续下去。当 receiver 程序接收到以 stop 开头的消息时就终止运行。

```
Response from server: stop
Sender has told me to end the connection
```

注意： sender.c 和 receiver.c 都不执行任何出错检查。这很不好但为了简单就这么做了。

试着改变地址看看发生了什么。试着改变端口再看看发生了什么。试着在运行 receiver 之前先运行 sender 再看看发生了什么。记住，你总能够通过使用 Ctrl+C 取消一个进程。

20.2.3 最少的出错检查

在最小程度上，你的代码应该核实套接口函数调用是否工作正确。毕竟有可能指定无效的 IP 地址或无效的端口，也有可能代码中函数的参数不正确。

核实函数调用是否能工作和核实数据本身是否实际传送到了不同。另外，我们可以通过使用主机名而不是 IP 地址来让代码更通用些。程序清单 20.3 所示的程序 sender2.c 使用主机名来取得 IP 地址，并且发送 20 个文本消息，然后发送终止消息（让接收方知道它已完成了）。

程序清单 20.3 sender2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
#include <netdb.h>
#include <errno.h>

int port = 6789;

void main() {
    int socket_descriptor;
    int iter = 0;
    ssize_t sendto_rc;
    int close_rc;
    char buf[80];
    struct sockaddr_in address;
    struct hostent *hostbyname;

/*
    Translate a host name to IP address
*/
    hostbyname = gethostbyname("127.0.0.1");
    if (hostbyname == NULL)
    {
        perror ("gethostbyname failed");
        exit (errno);
    }

/*
    Initialize socket address structure for Internet Protocols
    The address comes from the datastructure returned by
    gethostbyname()
*/
    bzero(&address, sizeof(address)); /* empty data structure */
    address.sin_family = AF_INET;
    memcpy(&address.sin_addr.s_addr, hostbyname->h_addr, sizeof
        (address.sin_addr.s_addr));
    address.sin_port = htons(port);

/*
    Create a UDP socket
*/
    socket_descriptor = socket(AF_INET, SOCK_DGRAM, 0);
    if (socket_descriptor == -1)
    {
        perror ("socket call failed");
        exit (errno);
    }

/*
    Loop 20 times (a nice round number) sending data.
*/
    for (iter = 0; iter <= 20; iter++)
    {
        sprintf(buf, "data packet with ID %d\n", iter);
```

```

        sendto_rc = sendto(socket_descriptor,
                           buf, sizeof(buf),
                           0, (struct sockaddr *)&address, sizeof(address));
    if (sendto_rc == -1)
    {
        perror ("sendto call failed");
        exit (errno);
    }
}
/*
Send a termination message
*/
sprintf(buf, "stop\n");
sendto_rc = sendto(socket_descriptor,
                   buf, sizeof(buf),
                   0, (struct sockaddr *)&address, sizeof(address));
if (sendto_rc == -1)
{
    perror ("sendto STOP call failed");
    exit (errno);
}
/*
Most people don't bother to check the return code
returned by the close function
*/
close_rc = close(socket_descriptor);
if (close_rc == -1)
{
    perror ("close call failed");
    exit (errno);
}

printf( "Messages Sent, Terminating\n");
exit (0);
}

```

程序 `sender2.c` 看上去和程序清单 20.1 中的 `sender.c` 很相像。这并不是偶然的。这两个程序除了在主机名翻译和增加错误检查之外是相同的。

函数 `gethostbyname` 对所给名字执行域名服务器查询并返回一个结构指针，这个结构包含了 `sockaddr_in` 结构所期望的那种形式的 IP 地址数据。

注意： `gethostbyname` 函数接受带点形式的 IP 地址或实际的主机名。

程序中使用了 `memcpy` 函数复制地址，而不是使用 `inet_addr` 函数把作为参数传递给它的字符串转换为整数值（以标准的带有点号的 IP 地址）。因为它已经是正确的格式了，所以不需要转换它。

在大多数情况下，要使用 `gethostbyname` 函数才行。

每个函数都返回一种不同的数据类型以及一种不同的出错提示。函数 `gethostbyname` 返回一个 NULL 指针, 其他函数返回 -1 的某些形式。在每个函数调用之后, 要检查返回代码。如果出现错误, 则使用 `perror` 函数显示确定出错位置的出错消息, 并且把出错代码翻译成文本。这可以让确定程序故障点的工作更容易些。

要编译这个程序, 应该使用

```
cc -o sender2 sender2.c
```

要运行它, 只需简单地输入 `sender2`。

如果发送方运行正常, 除了告诉你它已经结束的消息之外你不会看到其他任何消息。

这个程序传送原来的 `receiver` 程序能接收的数据。程序清单 20.4 所示的程序 `receiver2.c` 包含了有限的出错检查功能。

程序清单 20.4 `receiver2.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>

int port = 6789;

void main() {
    int sin_len;
    char message[256];
    int socket_descriptor;
    struct sockaddr_in sin;
    int bind_rc, close_rc;
    ssize_t recv_rc;

    printf("Waiting for data from sender\n");
    /*
     Initialize socket address structure for Internet Protocols
    */
    bzero(&sin, sizeof(sin)); /* empty data structure */
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(INADDR_ANY);
    sin.sin_port = htons(port);
    sin_len = sizeof(sin);
    /*
     Create a UDP socket
     and bind it to the port
    */
    socket_descriptor = socket(AF_INET, SOCK_DGRAM, 0);
    if (socket_descriptor == -1)
```



```
{
    perror ("socket call failed");
    exit (errno);
}

bind_rc = bind(socket_descriptor, (struct sockaddr *)&sin,
               sizeof(sin));
if (bind_rc == -1)
{
    perror ("bind call failed");
    exit (errno);
}
/*
Loop forever (or until a termination message is received)
Receive data through the socket and process it. The processing
in this program is really simple -- printing.
*/
while (1)
{
    recv_rc = recvfrom(socket_descriptor, message,
                       sizeof(message), 0,
                       (struct sockaddr *)&sin, &sin_len);
    if (recv_rc == -1)
    {
        perror ("recvfrom call failed");
        exit (errno);
    }

    printf("Response from server: %s\n", message);
    if (strncmp(message, "stop", 4) == 0)
    {
        printf("Sender has told me to end the connection\n");
        break;
    }
}
/*
Most people don't bother to check the return code
returned by the close function
*/
close_rc = close(socket_descriptor);
if (close_rc == -1)
{
    perror ("close call failed");
    exit (errno);
}

exit (0);
}
```

程序 `receiver2.c` 看上去和程序清单 20.2 中的 `receiver.c` 很相像。这并不是偶然的。这两个程序除了在简单的错误检查之外是相同的。

每个函数都返回一种不同的数据类型以及一种不同的出错提示。程序中的这些函数返回 -1 的某些形式。在每个函数调用之后, 要检查返回代码。如果出现错误, 则使用 `perror` 函数显示确定出错位置的出错消息并且把出错代码翻译成文本。这可以让确定程序故障点的工作更容易些。

要编译这个程序, 应该使用

```
cc -o receiver2 receiver2.c
```

要运行它, 只需简单地输入 `receiver2`。

在系统上测试这个程序最简单的办法就是打开两个命令提示窗口, 在不同的窗口中运行每个程序。首先要启动 `receiver2`, 因为 `sender2` 不会检查在另一端是否有人。在你运行 `receiver2` 的窗口中, 你会看到它的输出和 `receiver` 的输出非常相像。

注意: 因为端口和数据包的格式在 `sender`、`sender2`、`receiver` 以及 `receiver2` 之间是相同的, 你可以把任何一种发送程序和任何一种接收程序组合使用。

20.2.4 非阻塞 I/O

实际采用 UDP 的应用中接收方可能不会坐等发送方来的数据。它更有可能使用非阻塞 I/O, 周期性地检查数据, 并在数据到来之前进行其他处理工作。

现在程序本身必须能够处理等待。简单的应用可能会使用一种“忙等待”, 处于其中的进程检查数据, 当没有数据时就再检查, 直到有数据为止。这样做至少会浪费 CPU 资源。更好些的办法是使用能够让这个过程停止指定一段时间的函数, 比如使用标准 C 的 `sleep` 函数。最好的办法涉及信号和中断——这对于本章来说太复杂了。

当然, 这里假定了程序在持续等待数据到来的同时最好有什么事情要做。一个常见的实例是一个设备驱动程序等待连接并使用 `fork` 产生次级进程处理该连接。读者也可能想看看 `accept`、`select` 和 `fork` 的手册页面。

程序清单 20.5 所示的程序 `sender3.c` 对 `sender2.c` 做了微小修改, 加入了一个 `sleep` 函数来显示接收方检查数据的速度比发送数据的速度快。它发送 20 个文本消息, 然后发送一个终止消息 (让接收方知道它干完了)。

程序清单 20.5 `sender3.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>

int port = 6789;
```

```
void main() {
    int socket_descriptor;
    int iter = 0;
    ssize_t sendto_rc;
    int close_rc;
    char buf[80] ;
    struct sockaddr_in address;
    struct hostent *hostbyname;
    /*
     Translate a host name to IP address
    */
    hostbyname = gethostbyname("127.0.0.1");
    if (hostbyname == NULL)
    {
        perror ("gethostbyname failed");
        exit (errno);
    }
    /*
     Initialize socket address structure for
     Internet Protocols
     The address comes from the datastructure
     returned by gethostbyname()
    */
    bzero(&address, sizeof(address)); /* empty data structure */
    address.sin_family = AF_INET;
    memcpy(&address.sin_addr.s_addr,
        hostbyname->h_addr,
        sizeof(address.sin_addr.s_addr));
    address.sin_port = htons(port);
    /*
     Create a UDP socket
    */
    socket_descriptor = socket(AF_INET, SOCK_DGRAM, 0);
    if (socket_descriptor == -1)
    {
        perror ("socket call failed");
        exit (errno);
    }
    /*
     Loop 20 times (a nice round number) sending data.
    */
    for (iter = 0; iter <= 20; iter++)
    {
        sprintf(buf, "data packet with ID %d\n", iter);
        sendto_rc = sendto(socket_descriptor,
            buf, sizeof(buf),
```

```

        0, (struct sockaddr *)&address,
        sizeof(address));
    if (sendto_rc == -1)
    {
        perror ("sendto call failed");
        exit (errno);
    }
    sleep(3); /* this is the only difference from sender2.c */
}

/*
Send a termination message
*/

sprintf(buf, "stop\n");
sendto_rc = sendto(socket_descriptor,
    buf, sizeof(buf),
    0, (struct sockaddr *)&address, sizeof(address))
if (sendto_rc == -1)
{
    perror ("sendto STOP call failed");
    exit (errno);
}

/*
Most people don't bother to check the return code
returned by the close function
*/

close_rc = close(socket_descriptor);
if (close_rc == -1)
{
    perror ("close call failed");
    exit (errno);
}

printf ("Messages Sent, Terminating\n");
exit (0);
}

```

要编译这个程序, 应该使用

```
cc -o sender3 sender3.c
```

要运行它, 只需简单地输入 `sender3`。

如果发送方运行正常, 除了告诉你它已经结束的消息之外你不会看到其他任何消息。

程序清单 20.6 所示的程序 `receiver3.c` 包含了对 `receiver2` 的很大修改以处理非阻塞 I/O。一般说来, 它的行为还是和 `receiver2.c` 相似, 区别在于, 它不再等待发送方的数据——它能执行其他处理。在这个例子中, 它只打印没有数据的消息, 休息之后再尝试。

程序清单 20.6 receiver3.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>
#include <fcntl.h>

int port = 6789;

void main() {
    int sin_len;
    char message[256];
    int socket_descriptor;
    struct sockaddr_in sin;
    int bind_rc, close_rc;
    ssize_t recv_rc;
    long save_file_flags;

    printf("Waiting for data from sender\n");
    /*
     * Initialize socket address structure for Internet Protocols
     */
    bzero(&sin, sizeof(sin)); /* empty data structure */
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(INADDR_ANY);
    sin.sin_port = htons(port);
    sin_len = sizeof(sin);
    /*
     * Create a UDP socket
     * and bind it to the port
     */
    socket_descriptor = socket(AF_INET, SOCK_DGRAM, 0);
    if (socket_descriptor == -1)
    {
        perror("socket call failed");
        exit(errno);
    }

    bind_rc = bind(socket_descriptor, (struct sockaddr *)&sin,
sizeof(sin));
    if (bind_rc == -1)
    {
        perror("bind call failed");
        exit(errno);
    }
}
```

```

    }
/*
    set socket to non-blocking:
*/
    save_file_flags = fcntl(socket_descriptor, F_GETFL);
    printf("file flags are %ld\n", save_file_flags);
    save_file_flags |= O_NONBLOCK;
    if (fcntl(socket_descriptor, F_SETFL, save_file_flags) == -1)
    {
        perror("trying to set input socket to non-blocking");
        exit (errno);
    }
    printf("file flags are now %ld\n", save_file_flags);
/*
    Loop forever (or until a termination message is received)
    Receive data through the socket and process it. The processing
    in this program is really simple -- printing.
*/
    while (1)
    {
        sleep (1); /* wait a moment... */
        rcv_rc = recvfrom(socket_descriptor, message,
                          sizeof(message), 0, (struct sockaddr *)&sin, &sin_len);
        if (rcv_rc == -1 && errno != EAGAIN)
        {
            fprintf(stderr, "errno %d ", errno); perror ("recvfrom call
                                                         failed");
            exit (errno);
        }
        else if (rcv_rc == 0 | errno == EAGAIN) /* no data */
        {
            printf("no data yet\n");
            errno = 0; /* clear the error */
            continue;
        }

        errno = 0; /* clear the error */
        printf("Response from server: %s\n", message);
        if (strncmp(message, "stop", 4) == 0)
        {
            printf("Sender has told me to end the connection\n");
            break;
        }
    }
/*
    Most people don't bother to check the return code

```

```
        returned by the close function
    */
    close_rc = close(socket_descriptor);
    if (close_rc == -1)
    {
        perror ("close call failed");
        exit (errno);
    }

    exit (0);
}
```

receiver3.c 中最重要的新内容是 `fcntl` 调用，它允许对打开文件的控制标志做修改。带有参数 `F_GETFL` 的第一个调用用来取得当前控制标志的状态。然后 `O_NONBLOCK` 标志被添加到当前标志（通过使用按位“或”操作）中。带有参数 `F_SETFL` 的第二个调用实际上改变了标志。一旦这步成功完成，对套接口的读取（`recvfrom`）操作就不再等候数据了。当然，程序要核实 `fcntl` 的第二次调用是成功的。

`sleep` 函数将执行进程挂起指定的一段时间——此时为一秒钟。因为发送程序每隔 2 秒钟发送一个数据包，我们不希望程序不停地检查数据是否抵达。

程序一秒钟调用一次 `recvfrom`。根据调用产生的结果，采取不同的措施。如果出现一个错误，该函数就返回 -1 并且设置 `errno` 变量。如果因为资源不可用（`EAGAIN` 或者资源临时不可用）而没有取得数据，程序就显示一条消息来表明后续处理，同时错误被复位而循环会继续执行。如果是其他问题，则会引发正常的出错处理。如果接收到数据，处理该数据的方法和前面的例子中的方法相同。

另一种方法使用函数 `select`。这个函数等待输入，让它在套接口上可用。当然，当这个函数等待输入时，程序不会执行其他处理（此时只调用 `sleep`）。

在收到终止消息之前，`while` 循环一直在运行。每次收到一条消息，它就显示这条消息。消息本身非常简单——它们包含带有消息引用编号的文本。由单词 `stop` 构成的终止消息很重要，因为它的特殊含义为发送方和接收方共同理解。

实际应用可能会在执行一定次数的循环但没有收到数据的情况下做些什么。这种情况典型地叫做“超时”——接收方最终放弃等待发送方的数据并按顺序关闭。

要编译这个程序，你应该使用

```
cc -o receiver3 receiver3.c
```

要运行它，只需简单地输入 `receiver3`。

在系统上测试这个程序最简单的办法就是打开两个命令提示窗口，在不同的窗口中运行每个程序。首先要启动 `receiver3`，因为 `sender3` 不会检查在另一端是否有人。在你运行 `receiver3` 的窗口中，你会看到类似这样的输出：

```
Waiting for data from sender
file flags are 2
file flags are now 130
no data yet
```

```
no data yet
no data yet
no data yet
no data yet
Response from server: data packet with ID 0

no data yet
not data yet
Response from server: data packet with ID 1

no data yet
not data yet
Response from server: data packet with ID 2

no data yet
not data yet
Response from server: data packet with ID 3

no data yet
no data yet
```

只要发送方发出的数据包到达接收方, 就会持续显示这样的输出。当 receiver3 接收到以 stop 开头的消息时才终止执行:

```
Response from server: stop
Sender has told me to end the connection
```

注意: 你可能会看到文件标志的不同值。这个特殊的值是实现特定的, 但是符号链接的名字 (比如 O_NONBLOCK) 是标准的。

当 receiver3 正在运行的时候, 你甚至可以杀死 sender3 然后再重新启动 sender3。试试吧。

20.3 小 结

正如本章中的示例程序所表明的那样, 创建并使用 UDP 套接口相当简单。重要的是记住 UDP 不保证数据包会抵达它的目的地, 也不保证在目的地会有程序等待接收数据包。你也应该记住, 在接收方程序不知情的情况下, 发送方程序可以消失 (崩溃或被杀死)。

当你在局域网上工作时, 不太可能丢失 UDP 消息数据包, 所以最少量的握手机制常常就足够能保证发送方和接收方程序正确运行了。

如果你要保证数据传输正确而且发送方和接收方程序能够很容易地检测到对方的故障, 就该使用 TCP。选择什么协议取决于你的应用程序。

第 21 章 多播套接口和非阻塞 I/O

多播广播是用于建立分布式系统，比如聊天工具、公用黑板绘画工具以及远程视频会议系统的一项非常好的工具。使用多播广播的程序和使用 UDP 向单个接收方发送消息的程序非常相似。最主要的区别在于多播广播程序中使用特殊的多播 IP 地址。例如，本地计算机的 IP 地址是 127.0.0.1，而它的多播 IP 地址是 224.0.0.1。读者将会看到接收多播广播的程序和接收一般 UDP 消息的程序有很大的差别。

为什么多播有特殊的 IP 地址呢？回想起标准的 IP 地址被分成三类——A、B 和 C——构成了地址范围。而多播 IP 地址是 D 类地址。RFC 1390 定义了多播广播地址是 224.0.0.1。

并不是所有的计算机都被配置成 IP 多播方式。本章你将学到如何配置 Linux 来支持 IP 多播方式。如果你使用的是异构计算机网络，则要知道 Windows NT 支持多播 IP，但是 Windows 95 则需要一个免费的补丁。大部分现代的网卡都支持 IP 多播。即使你的网卡不支持 IP 多播（这种情况不太可能），本章中的示例程序仍然能够建立并在单机上运行，因此你仍然能够试验多播 IP 编程。

21.1 配置 Linux 支持多播 IP

在默认状态下，大多数 Linux 系统都关闭了对多播 IP 的支持。为了在你的 Linux 系统上使用多播套接口，需要重新配置并编译你的内核。

提示： 为了配置你的内核以支持多播，一种方法是，如果存在 `/usr/src/linux/.config` 这个文件，则要编辑它，确保 `CONFIG_IP_MULTICAST` 一行不被注释而且设置为 `y`。也就是说，要保证在这个文件中有类似下面的一行：

```
CONFIG_IP_MULTICAST=y
```

当然，你也可以使用标准的内核配置工具完成上述工作。

一旦你有了支持 IP 多播的内核，以超级用户身份执行下面的命令：

```
# router add -net 224.0.0.0 netmask 224.0.0.0 dev lo
```

要核实这条路由配置已经加入到系统了，可输入：

```
# route -e
```

你应该能看到类似下面的输出：

```
# route -e
Kernel IP routing table
Destination      Gateway          Genmask          Flags MSS Window  irtt  Iface
```

```

10.0.0.0      *      255.255.255.0 U      0 0      0 eth0
127.0.0.0      *      255.0.0.0   U      0 0      0 lo
BASE-ADDRESS.MC *      240.0.0.0   U      0 0      0 lo
default      10.0.0.1  0.0.0.0   UG     0 0      0 eth0
#

```

请注意，我是以超级用户身份运行 `route` 命令的。我并不打算把这条多播路由永久地加入到我的 Linux 系统中；相反，只有当我需要使用多播 IP 的时候才手动地将其加入（以超级用户身份）。

21.2 为支持多播 IP 重新编译 Linux 内核

为支持 IP 多播而重新配置并编译内核的工作是相当简单的。采用下面的步骤配置并编译支持多播 IP 的新内核。

1. `cd /usr/src/linux`
2. `make menuconfig`
3. 选择网络选项
4. 选中 IP: Enable Multicasting IP 一项
5. 保存并从 `menuconfig` 退出
6. `make dep; make clean; make bzImage`
7. `cp /vmlinuz /vmlinuz_good`
8. `cp arch/i386/boot/zImage /vmlinuz`
9. `cd /etc`
10. 编辑 `lilo.conf`, 加入针对 `/vmlinuz_good` 内核的新项
11. `lilo`

提示： 请阅读能得到的针对编译和安装新 Linux 内核的文档。Linux 带有很多 HOWTO 形式的联机文档。在你开始工作之前请先阅读有关编译新内核并配置支持多播 IP 的 HOWTO 文档。

21.3 多播 IP 广播的示例程序

支持多播 IP 方式并未用到新的系统 API 调用；相反，对现有函数 `getsockopt` 和 `setsockopt` 进行扩展，使之带有支持多播 IP 的选项即可。这些函数的原型如下：

```

#include <sys/types.h>
#include <sys/socket.h>
int getsockopt (int socket, int level, int optname,
               void *optval, int *optlen);
int setsockopt (int socket, int level, int optname,

```

```
const void *optval, int optlen);
```

若需要看关于两个函数的完整的文档，请使用命令 `man getsockopt`。以下的讨论仅限于说明建立套接口以支持多播 IP 时函数 `getsockopt` 和 `setsockopt` 的用法。第二个参数 `level` 设置被选中的操作协议级别。在例子程序中我们一直将变量 `level` 指定为 `IPPROTO_IP`。第三个参数 `optname` 是一个代表可能选项的整型值。用于多播 IP 的选项为：

- `SO_REUSEADDR`——使一个给定的地址可以在同一台计算机上同时被一个以上的进程使用
- `SO_BROADCAST`——使多播广播有效
- `IP_ADD_MEMBERSHIP`——通知 Linux 内核此套接口将被用于多播
- `IP_DROP_MEMBERSHIP`——通知 Linux 内核此套接口不再参与多播 IP
- `IP_MULTICAST_TTL`——指定广播消息的生存时间值
- `IP_MULTICAST_LOOP`——指定信息是否被广播到发送计算机：默认值为真，因此当运行在你的 Linux 计算机上的程序广播消息时，同样运行在你的计算机上的其他程序也会收到消息

你将看到在例子 `broadcast.c` 和 `listen.c` 中设置这些值的一些例子。程序会一直检查这些函数的错误返回值。错误返回值是小于 0 的值。通常检查小于 0 的返回值并调用系统函数 `perror` 输出最近的系统错误就足够了。但是你可能要检查你程序中的具体错误，可以使用以下定义的常量来显示具体的错误代码：

- `EBADF`——不正确的套接口描述字（第一个参数）
- `ENOTSOCK`——`socket`（第一个参数）是一个文件而不是套接口
- `ENOPROTOOPT`——在指定的协议层上未知的选项
- `EFAULT`——`optval`（第四个参数）引用的地址不在当前进程的地址空间

21.3.1 使用多播 IP 广播数据

使用多播 IP 进行广播看上去和使用 UDP 发送数据（参见第 20 章“UDP：用户数据报协议”）类似。本节中示例程序的源代码保存在本章源代码目录 `IPC/MULTICAST` 下的文件 `broadcast.c` 里。这个文件如程序清单 21.1 所示。通过执行 `make broadcast` 使用本书提供的 `makefile` 文件编译这个程序。

程序清单 21.1 `broadcast.c`

```
/*
 * broadcast.c - An IP multicast server
 *
 * Copyright Mark Watson 1999. Open Source Software License.
 * Hacked on by Kurt Wall
 */

#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```

#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <stdlib.h>

int port = 6789;

int main(void)
{
    int socket_descriptor;
    struct sockaddr_in address;

    /*
     * For broadcasting, this socket can be treated
     * like a UDP socket:
     */
    socket_descriptor = socket(AF_INET, SOCK_DGRAM, 0);
    if (socket_descriptor == -1) {
        perror("Opening socket");
        exit(EXIT_FAILURE);
    }

    /* Initialize the address structure */
    memset(&address, 0, sizeof(address));
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr("224.0.0.1");
    address.sin_port = htons(port);

    /* Start broadcasting */
    while(1) {
        if(sendto(socket_descriptor, "test from broadcast",
            sizeof("test from broadcast"), 0,
            (struct sockaddr *)&address, sizeof(address)) < 0) {
            perror("sendto");
            exit(EXIT_FAILURE);
        }
        sleep(2);
    }

    exit(EXIT_SUCCESS);
}

```

首先要按常规设立套接口:

```

int socket_descriptor;
struct sockaddr_in address;
socket_descriptor = socket(AF_INET, SOCK_DGRAM, 0)
if (socket_descriptor == -1) {
    perror ("Opening socket");
    exit(1);
}

```

注意，和前面章节中 UDP 的例子类似，`socket` 调用的第二个参数为 `SOCK_DGRAM`，它指定了一个无连接的套接口。下一段代码使用 `memset` 初始化地址数据结构，并且设置数据结构能够用 UDP 套接口发送数据：

```
memset(&address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_addr.s_addr = inet_addr("224.0.0.1");
address.sin_port = htons(6789); // we are using port 6789
```

在此不同寻常的一点是 IP 地址 224.0.0.1 是本地 IP 地址 127.0.0.1 的多播等价地址。无限 while 循环每 2 秒把字符串 `test from broadcast` 广播出去：

```
while (1) {
    if (sendto(socket_descriptor,
               "test from broadcast", sizeof("test from broadcast"),
               0, (struct sockaddr *)&address, sizeof(address))
        < 0) {
        perror("Trying to broadcast with sendto");
        exit(1);
    }
    sleep(2);
}
```

除了多播 IP 地址 224.0.0.1 之外，`broadcast.c` 和前面章节中使用 UDP 发送数据的示例程序相同。

21.3.2 创建客户程序监听多播 IP 广播

一旦你建好了广播服务器，就需要有一个客户端程序收听多播广播。收听多播 IP 广播消息的示例程序和 UDP 的示例程序大相径庭。收听多播广播要求你在这个程序中做几项新工作：

- 多播收听方，或者说客户端必须通知 Linux 内核某个指定的套接口将加入多播 IP 广播组。
- 收听方必须让运行在同一计算机的不同进程共用一个套接口。
- 必须配置套接口使得广播消息可以发送给同一主机（这是默认行为）。这样就可以在同一台计算机上测试广播程序和收听程序的多个副本。

程序清单 21.2 显示了 `listen.c` 的源代码。提供执行 `make listen` 使用本书提供的 `makefile` 文件编译这个程序。

程序清单 21.2 `listen.c`

```
/*
 * listen.c - An IP multicast client
 *
 * Copyright Mark Watson 1999. Open Source Software License.
 * Hacked on by Kurt Wall
```

```
*/

#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdlib.h>
#include <unistd.h>

char * host_name = "224.0.0.1"; /* Address for multicast IP */
int port = 6789;

int main(void)
{
    struct ip_mreq command;
    int loop = 1; /* The broadcast loops back to localhost */
    int iter = 0;
    int sin_len;
    char message[256];
    int socket_descriptor;
    struct sockaddr_in sin;
    struct hostent *server_host_name;

    if((server_host_name = gethostbyname(host_name)) == 0) {
        perror("gethostbyname");
        exit(EXIT_FAILURE);
    }

    bzero(&sin, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(INADDR_ANY);
    sin.sin_port = htons(port);

    if((socket_descriptor = socket(PF_INET, SOCK_DGRAM, 0)) == -1)
    {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    /* Allow multiple processes to use this same port:*/
    loop = 1;
    if(setsockopt(socket_descriptor, SOL_SOCKET, SO_REUSEADDR,
        &loop, sizeof(loop)) < 0) {
        perror("setsockopt:SO_REUSEADDR");
        exit(EXIT_FAILURE);
    }

    if(bind(socket_descriptor, (struct sockaddr *)&sin, sizeof(sin))
        < 0) {
        perror("bind");
    }
}
```

```

        exit(EXIT_FAILURE);
    }

    /* Allow broadcast to this machine */
    loop = 1 ;
    if(setsockopt(socket_descriptor, IPPROTO_IP, IP_MULTICAST_LOOP,
        &loop, sizeof(loop)) < 0) {
        perror(" setsockopt :IP_MULTICAST_LOOP");
        exit(EXIT_FAILURE);
    }

    /* Join the broadcast group:*/
    command.imr_multiaddr.s_addr = inet_addr("224.0.0.1");
    command.imr_interface.s_addr = htonl(INADDR_ANY);

    if (command.imr_multiaddr.s_addr == -1) {
        perror("224,0.0.1 not a legal multicast address");
        exit(EXIT_FAILURE);
    }

    if (setsockopt(socket_descriptor, IPPROTO_IP,
        IP_ADD_MEMBERSHIP, &command, sizeof(command)) < 0) {
        perror("setsockopt:IP_ADD_MEMBERSHIP");
    }

    while(iter++ < 10) {
        sin_len = sizeof(sin);
        if(recvfrom(socket_descriptor, message, 256, 0,
            (struct sockaddr *)&sin, &sin_len) == -1) {
            perror("recvfrom") ;
        }
        printf("Response %d-2d from server: %s\n", iter, message);
        sleep(2);
    }

    /* leave the broadcast group */
    if (setsockopt (socket_descriptor, IPPROTO_IP,
        IP_DROP_MEMBERSHIP, &command, sizeof(command)) < 0) {
        perror("setsockopt:IP_DROP_MEMBERSHIP");
    }

    close(socket_descriptor) ;

    exit(EXIT_SUCCESS);
}

/*To set up the socket */
int socket_descriptor;
struct sockaddr_in sin;
struct hostent *server host name
if ((server_host_name = gethostbyname(host_name)) == 0) {
    perror("Error resolving local host\n");
}

```

```

        exit(1);
    }

    if ((socket_descriptor = socket(PF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("Error opening socket\n");
        exit(1);
    }

```

前三段代码按通常方式设置套接口。但是在调用 `bind` 之前，你必须设置套接口的选项启用多播 IP 支持。首先，使用 `setsockopt` 调用让多个进程共享同一端口：

```

/* Allow multiple processes to use this same port:*/
loop = 1 ;
if(setsockopt(socket_descriptor, SOL_SOCKET, SO_REUSEADDR,
    &loop, sizeof(loop)) < 0) {
    perror("setsockopt:SO_REUSEADDR");
    exit(EXIT_FAILURE);
}

```

既然已经配置套接口为共享使用，就可以用 `bind` 调用把它绑定到一个端口。下一步为在同一主机上进行广播而设置套接口——这样做能够方便地在单个开发系统上测试多播 IP 广播：

```

/* Allow broadcast to this machine */
loop = 1;
if(setsockopt(socket_descriptor, IPPROTO_IP, IP_MULTICAST_LOOP,
    &loop, sizeof(loop)) < 0 ) {
    perror ("setsockopt:IP_MULTICAST_LOOP");
    exit(EXIT_FAILURE);
}

```

下一步是加入一个广播组。这一步告诉 Linux 内核，特定套接口即将到来的数据是广播数据：

```

/* Join the broadcast group:*/
command.imr_multiaddr.s_addr = inet_addr("224.0.0.1");
command.imr_interface.s_addr = htonl(INADDR_ANY);

if (command.imr_multiaddr.s_addr == -1) {
    perror("224.0.0.1 not a legal multicast address");
    exit(EXIT_FAILURE);
}

if(setsockopt(socket_descriptor, IPPROTO_IP, IP_ADD_MEMBERSHIP,
    &command, sizeof(command)) < 0) {
    perror("setsockopt:IP_ADD_MEMBERSHIP");
}

```

此时，已经为多播 IP 配置好了套接口。下面要做的就是进入一个无限循环，在多播套接口上收听广播：


```
While(iter++ < 10) {
    sin_len = sizeof(sin);
    if(recvfrom(socket_descriptor,message,256,0,
        (struct sockaddr *)&sin, &sin_len) == -1) {
        perror("recvfrom");
    }
    printf("Response #%-2d from server: %s\n",iter,message);
    sleep(2);
}
```

注意，收听方只收听 10 个广播消息，然后就退出。此时，收听方离开广播组并且关闭套接口：

```
if (setsockopt(socket_descriptor, IPPROTO_IP, IP_DROP_MEMBERSHIP,
    &command, sizeof(command)) < 0) {
    perror("Error in setsocket(drop membership)");
}
close(socket_descriptor);
```

21.3.3 运行多播 IP 示例程序

要运行示例程序，打开两个终端窗口并在每个窗口中进入本章的源代码目录（或者你构建程序的任何位置）。在一个窗口中键入 `make`，编译 `broadcast` 和 `listen` 的可执行文件。在另一个窗口中执行 `./broadcast`，启动服务器程序。在另一个窗口中通过执行 `./listen`，启动收听方程序。在收听方的窗口中，你应该看到如下输出：

```
$ ./listen
Response #1 from server: test from broadcast
Response #2 from server: test from broadcast
Response #3 from server: test from broadcast
Response #4 from server: test from broadcast
Response #5 from server: test from broadcast
Response #6 from server: test from broadcast
Response #7 from server: test from broadcast
Response #8 from server: test from broadcast
Response #9 from server: test from broadcast
Response #10 from server: test from broadcast
```

而广播窗口应该显示：

```
$ ./broadcast
```

注意，两个程序都用普通用户帐号而不是超级用户帐号来执行。只有当构建一个新内核以及执行 `route` 命令时才需要超级用户身份。你必须通过在窗口中按 `Ctrl+C` 键来杀死服务器程序 `broadcast`。

21.4 小 结

多播 IP 是一种同时向几个进程高效地广播信息的良好技术。在本章里，你学到了广播多播 IP 程序几乎和使用 UDP 通过套接口发送数据的程序相同，但是为接收多播 IP 广播你必须完成额外的一些工作。多播 IP 可用于连网游戏以及广播音频和视频数据的程序。

第 5 部分 用户界面编程

第 22 章 底层终端控制

本章介绍在 Linux 下任何对终端进行控制。这一章将向你展示用于控制终端以及用于 Linux 应用屏幕输出的低层 API。从使用计算机进行计算工作的最早时期开始，当时用户还只是从哑终端和 CPU 进行交互，终端控制便一直是一种古老而又重要的概念。虽然这一概念非常古老，但是它的基础思想仍然定义着 Linux（和 UNIX）与大多数输入、输出设备进行通信的方式。

22.1 终端接口

终端接口，即 tty 接口是从用户只能坐在连接到一台打印机的一个比打字机稍大些的键盘前工作的时期演化而来的。这个键盘就是输入设备，而打印机就是屏幕，打印机回写输入和输出。tty 接口基于的硬件模型假设键盘和打印机的组合是通过串口连接到一台远程计算机系统上。这一模型当前流行的客户机/服务器计算体系结构有一定的渊源。图 22.1 显示了这一硬件模型。

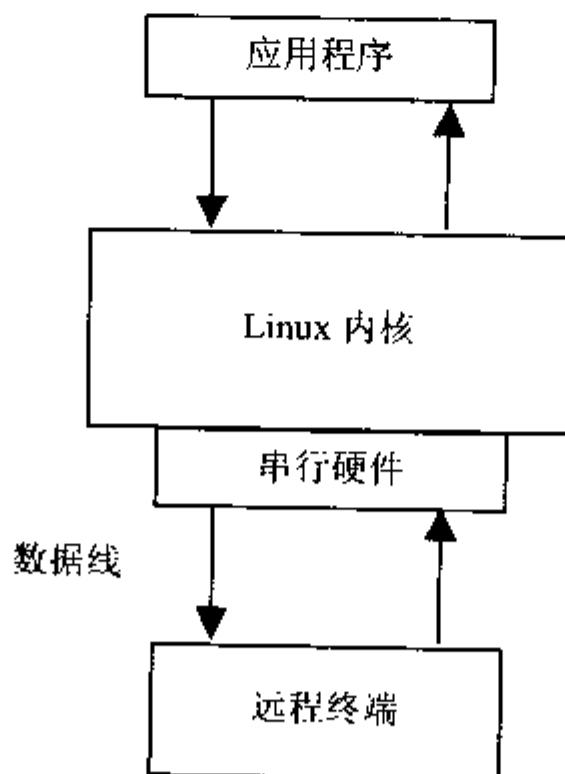


图 22.1 终端硬件模型

虽然上述模型很复杂，令人望而生畏。但该模型非常通用，以致于几乎在任何一个需要将程序与某种输入、输出设备（比如打印机、控制台、x 终端和网络登录等）打交道的

情况下都可以被描述为这种通用模型的子类型。结果，因为这一模型提供了一个一致的，可以被应用于许多种不同情况的编程接口，它实际上简化了程序员的任务。

终端接口为什么会如此复杂？原因也许有很多，但其中最重要的两个原因是人本身的因素和终端接口必须要完成的任务。除了必须管理用户和系统之间、程序和系统之间以及设备和系统之间的交互之外，终端接口还不得不对几乎无限种类的来源完成接收输入和发送输出的操作。考虑到所有不同的键盘类型、鼠标、游戏杆以及其他用来传输用户输入数据的设备，再考虑所有这些不同类型的输出设备，像调制解调器、打印机、绘图仪、串行设备、视频卡以及显示器，终端接口必须能够适应所有这些类型的设备。硬件过多产生了一定的复杂性。

复杂性同时也会由人本身的因素产生。因为终端是交互的，用户（人）就会想要控制这种交互，并按照他们的习惯和喜好进行改进。随着终端大小和特性集的增长，作为这个人的因素的一个直接结果就是复杂性产生相应的增长。

下面将告诉你如何使用 POSIX 的 `termios` 接口来控制终端的行为。`termios` 提供了对 Linux 如何接收和处理输入的细粒度控制。在本章后面“使用 `terminfo`”中，你将学习如何控制在控制台或者 `xterm` 显示屏输出的外观。

22.2 控制终端

POSIX.1 定义了一个查询和操纵终端的标准接口，该接口被称作 `termios`，在系统的头文件 `<termios.h>` 中定义。`termios` 非常类似 System V UNIX 的 `termio` 模型，但也加入了一些由 Berkeley UNIX 衍生出的 UNIX 系统的一些终端接口特性。从程序员的角度来看，`termios` 是一个数据结构和一系列操纵这些数据结构的函数。下面列出了 `termios` 的数据结构，它包含了一个终端特性的完整描述，相关联的函数可以查询和修改这些特性。

```
#include <termios.h>

struct termios {
    tcflag_t c_iflag;        /* input mode flags */
    tcflag_t c_oflag;        /* output mode flags */
    tcflag_t c_cflag;        /* control mode flags */
    tcflag_t c_lflag;        /* local mode flags */
    cc_t c_line;             /* line discipline */
    cc_t c_cc[NCCS];         /* control characters */
    speed_t c_ispeed;        /* input speed */
    speed_t c_ospeed;        /* output speed */
};
```

其中的 `c_iflag` 成员是用来控制输入处理选项的，它将影响到终端驱动程序在把输入发送给程序前是否对其进行处理，及怎样对其进行处理。`c_oflag` 成员用来控制输出数据的处理，并决定在发送输出数据到显示屏和其他输出设备之前，终端驱动程序是否以及如何来处理它们。各种决定终端设备硬件特性的控制标志是在 `c_cflag` 成员中设置的。存放在 `c_lflag`

中的本地模式标志用来操纵这样的一些终端特性，比如是否将输入字符显示到显示屏上。数组 `c_cc` 包含了特殊字符序列的值，比如 `^\\`（退出）和 `^H`（删除），以及它们所代表的操作。成员 `c_line` 表示控制协议，比如 SLIP、PPP，或者 X.25——我们这里不讨论该成员，因为它的使用已经超出了本书的范围。

终端有两种工作模式，分别为规范模式（或称为 `cooked` 模式）和非规范模式（或称为原始模式）。在规范模式下，终端设备驱动程序处理特殊字符并以一次一行的方式将输入发送给程序使用；而在非规范模式下，大多数键盘输入将得不到处理，也不缓存。命令解释器程序 `shell` 就是一个使用规范模式的应用程序例子。而另一方面，全屏编辑器 `vi` 则使用非规范模式。`vi` 在输入数据键入的同时接收数据，并且自己处理大多数的特殊字符（例如 `^D`，在 `vi` 中将光标移动到文件的末尾，但只给 `shell` 发送一个 EOF 信号）。

表 22.1 列出了常用的终端控制模式标志。

表 22.1 POSIX `termios` 标志

标志	成员	描述
IGNBRK	<code>c_iflag</code>	忽略输入中的 BREAK 条件
BRKINT	<code>c_iflag</code>	如果设置了 IGNBRK，将在 BREAK 时产生 SIGINT
INLCR	<code>c_iflag</code>	将输入中的 CR 转换成 NL
IGNCR	<code>c_iflag</code>	忽略输入中的 CR
ICRNL	<code>c_iflag</code>	如果没有设置 IGNCR，将输出中的 CR 转换为 NL
ONLCR	<code>c_oflag</code>	将输出中的 NL 映射为 CR-NL
OCRNL	<code>c_oflag</code>	将输出中的 CR 映射为 NL
ONLRET	<code>c_oflag</code>	不输出 CR
HUPCL	<code>c_cflag</code>	在最后处理结束时关闭设备并关闭连接
CLOCAL	<code>c_cflag</code>	忽略调制解调器的控制线
ISIG	<code>c_lflag</code>	当分别有 INTR、QUIT 或者 SUSP 字符被接收时，将产生 SIGINT、SIGQUIT 和 SIGSTP
ICANON	<code>c_lflag</code>	启用规范模式
ECHO	<code>c_lflag</code>	将输入字符显示到输出中
ECHONL	<code>c_lflag</code>	在规范模式中，即使没有设置 ECHO 也显示 NL 字符

控制字符数组 `c_cc` 至少包含了 11 种特殊控制字符，例如 `^D` EOF、`^C` INTR 和 `^U` KILL。这 11 个特殊控制字符中，有 9 个可以被改变。而 CR（回车符）总是 `\r`，NL（换行符）总是 `\n`；这两个是不能改变的。

22.2.1 属性控制函数

`termios` 接口包含了许多控制终端特性的函数。其中基本的函数包括 `tcgetattr` 和 `tcsetattr`。`tcgetattr` 用来初始化一个 `termios` 数据结构，并设置用来表示该终端特性和设置的属性值。查询和改变这些设置即是使用下面章节中将讨论的函数来操纵由 `tcgetattr` 返回的数据结构。一旦完成了这些操作，使用 `tcsetattr` 用得到的新值来更新终端。下面将列出 `tcgetattr` 和 `tcsetattr` 的函数原型，并对它们进行解释。

```
#include <termios.h>
#include <unistd.h>
int tcgetattr(int fd, struct termios *tp);
```

tcgetattr 查询和文件描述符 **fd** 相关联的终端参数，并将它们存储到由 **tp** 所引用的 **termios** 结构中。如果成功，则返回 0；发生错误则返回 -1。

```
#include <termios.h>
#include <unistd.h>
int tcsetattr(int fd, int action, struct termios *tp);
```

tcsetattr 使用由 **tp** 引用的 **termios** 结构来设置和文件描述符 **fd** 相关联的终端参数。参数 **action** 通过使用下面的参数值来控制什么时候改变生效。

- **TCSANOW**——立即改变这些属性值。
- **TCSADRAIN**——改变发生在当所有 **fd** 上的输出都已经被发送到终端以后。当要改变输出设置时使用此函数。
- **TCSAFLUSH**——改变发生在所有 **fd** 上的输出都已经被发送到终端以后，但任何挂起的输入将被丢弃。

22.2.2 速度控制函数

前四个函数用来设置终端设备的输入、输出速度。因为该接口已经出现很长时间了，所以它仍以波特率来定义速度，虽然正确的术语应该是比特每秒 (bps)。这些函数都是成双出现，两个用来获取和设置输出线路的速度，两个用来获取和设置输入线路的速度。它们的函数原型在 **<termios.h>** 头文件中定义。下面列出了这些函数的原型和它们执行操作的简短描述。

```
#include <termios.h>
#include <unistd.h>
int cfgetispeed(struct termios *tp);
```

cfgetispeed 返回由 **tp** 指针指向的 **termios** 结构中所存储的输入线路的速度值。

```
#include <termios.h>
#include <unistd.h>
int cfsetispeed(struct termios *tp, speed_t speed);
```

cfsetispeed 将由 **tp** 指针指向的 **termios** 结构中存储的输入线路速度值设置为 **speed**。

```
#include <termios.h>
#include <unistd.h>
int cfgetospeed(struct termios *tp);
```

cfgetospeed 返回由 **tp** 指针指向的 **termios** 结构中所存储的输出线路的速度值。

```
#include <termios.h>
#include <unistd.h>
int cfsetospeed(struct termios *tp, speed_t speed);
```

`cfsetospeed` 将由 `tp` 指针指向的 `termios` 结构中存储的输出线路速度值设置为 `speed`。参数 `speed` 必须是下列常数之一：

B0（关闭连接）	B1800
B50	B2400
B75	B4800
B110	B9600
B134	B19200
B150	B38400
B200	B57600
B300	B115200
B600	B230400

22.2.3 行控制函数

行控制函数用来查询和设置各种与数据如何、什么时候以及是否流向终端设备相关的特征。这些函数使你能够对终端设备的行为进行一种合适量度的控制。例如，为了在继续向前进行下去之前，使所有挂起的输出操作完成，可以使用 `tcdrain` 函数，其函数原型为：

```
#include <termios.h>
#include <unistd.h>
int tcdrain(int fd);
```

`tcdrain` 将一直保持等待，直到所有的输出都已经写到文件描述符 `fd` 指向的文件为止。为了强迫输出、输入或者两者同时刷新，可以使用 `tcflush` 函数。它的原型如下所示：

```
#include <termios.h>
#include <unistd.h>
int tcflush(int fd, int queue);
```

`tcflush` 刷新排在文件描述符 `fd` 的队列中的输入和输出（或两者都刷新）。`Queue` 参数就是用来指定要刷新的数据的，如下面所示：

- `TCIFLUSH`——刷新接收到但尚未读取的输入数据。
- `TCOFLUSH`——刷新被改写但尚未传送的输出数据。
- `TCIOFLUSH`——刷新接收到但尚未读取的输入数据和已写入但尚未传送的输出数据。

实际的流量控制，不管是处于开状态还是关状态，都由 `tcflow` 函数来控制，其函数原型如下所示：

```
#include <termios.h>
#include <unistd.h>
int tcflow(int fd, int action);
```

`tcflow` 按照 `action` 参数的不同取值来启动或停止对文件描述符 `fd` 的数据传送和接收：

- TCOON——启动输出
- TCOOFF——停止输出
- TCION——启动输入
- TCIOFF——停止输入

22.2.4 进程控制函数

`termios` 接口定义的进程控制函数使你能够得到关于在某个给定终端上运行进程（程序）的信息。得到这一信息的关键是进程组。例如，为了找到进程组在某个给定终端上的标识号，可以使用函数 `tcgetpgrp`，其原型如下所示：

```
#include <termios.h>
#include <unistd.h>
pid_t tcgetpgrp(int fd);
```

`tcgetpgrp` 返回在打开文件描述符为 `fd` 的终端上前台运行的进程组 `pgrp_id` 的进程组标识号，如果发生错误则返回-1。

如果你的程序有足够的访问权限（与 `root` 相当的权限），使用 `tcsetpgrp` 函数可以改变进程组的标识符。`tcsetpgrp` 的函数原型如下所示：

```
#include <termios.h>
#include <unistd.h>
int_t tcsetpgrp(int fd, pid_t pgrp_id);
```

`tcsetpgrp` 在打开文件描述符为 `fd` 的终端上设置前台进程组标识符为进程组标识 `pgrp_id`。

除非特别说明，所有的函数在调用成功后都返回 0。在发生错误时，这些函数返回-1，并设置 `errno` 以表明错误类型。图 22.2 描述了图 22.1 表示的硬件模型和 `termios` 数据结构之间的关系。

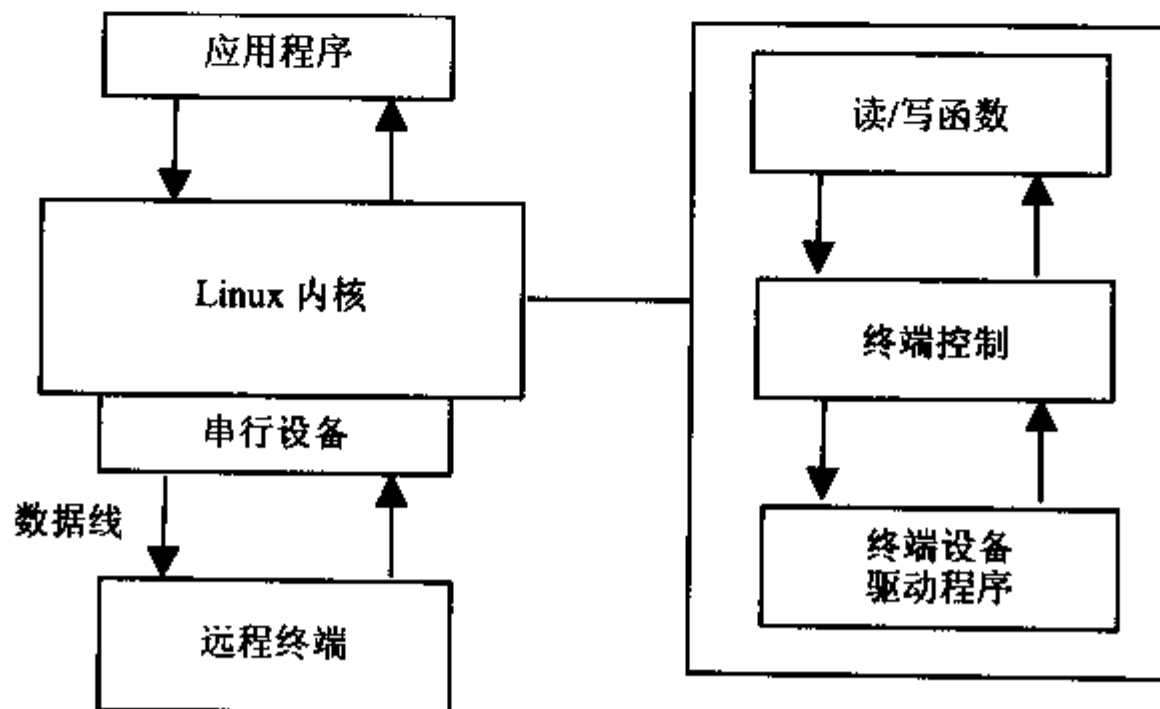


图 22.2 硬件模型到 `termios` 数据结构的映射方式

`c_lflag`、`c_iflag`、`c_oflag`、`c_cflag` 这四个标志，在终端设备驱动程序和程序的输入、

输出函数之间进行协调输入。在内核和用户程序之间的读函数和写函数是用户程序空间和核心的接口。这些函数可以是简单的 `fgets` 和 `fputs`，可以是 `read` 和 `write` 系统调用，或者是其他的输入输出函数，这些都取决于程序本身的性质。

22.3 使用终端接口

在所有操纵 `termios` 数据结构的函数中，最常用的是 `tcgetattr` 和 `tcsetattr` 两个函数。顾名思义，`tcgetattr` 函数查询一个终端的状态，而 `tcsetattr` 函数用来改变一个终端的状态。这两个函数都接受一个对应该进程的控制终端的文件描述符 `fd` 和一个指向 `termios` 结构的指针。正如上面所述的那样，`tcgetattr` 获得被引用的 `termios` 结构，而 `tcsetattr` 使用存储在该结构中的属性值来更新终端的各种属性。

程序清单 22.1 中显示了如何使用 `termios` 在输入口令字时关闭字符回显，这也是 `termios` 接口一种常见的用途。

程序清单 22.1 `noecho.c`

```
/*
 * noecho.c - Using termios to disable key echo
 */
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>

#define PASS_LEN 8

void err_quit(char *msg, struct termios flags);

int main(void)
{
    struct termios old_flags, new_flags;
    char password[PASS_LEN + 1];
    int retval;

    /* Get the current terminal settings */
    tcgetattr(fileno(stdin), &old_flags);
    new_flags = old_flags;

    /* Turn off local echo, but pass the newlines through */
    new_flags.c_lflag &= -ECHO;
    new_flags.c_lflag |= ECHONL;

    /* Did it work? */
    retval = tcsetattr(fileno(stdin), TCSAFLUSH, &new_flags);
    if (retval != 0)
        err_quit("Failed to set attributes", old_flags);

    /* Did the settings change? */
    tcgetattr(fileno(stdin), &new_flags);
```

```

    if(new_flags.c_lflag & ECHO) {
        err_quit("Failed to turn off ECHO", old_flags);
    }
    if(!new_flags.c_lflag & ECHONL) {
        err_quit("Failed to turn on ECHONL", old_flags);
    }

    printf("Enter password: ");
    fgets(password, PASS_LEN + 1, stdin);
    printf("You typed: %s", password);

    /* Restore the old termios settings */
    tcsetattr(fileno(stdin), TCSANOW, &old_flags);

    exit(EXIT_SUCCESS);
}

void err_quit(char *msg, struct termios flags)
{
    fprintf(stderr, "%s\n", msg);
    tcsetattr(fileno(stdin), TCSANOW, &flags);
    exit(EXIT_FAILURE);
}

```

在各种变量的定义之后，第一段代码检索 `stdin` 当前的 `termios` 设置状态。通过把当前终端的设置复制到 `new_flags` 结构中，你可以在不丢失原来设置的情况下操纵它们。一个表现良好的程序应该在退出之前恢复原来的设置，所以在这个程序退出之前，已恢复了原来的 `termios` 设置。

第二段代码演示了清除和设置 `termios` 标志的正确方法。第一行关闭本地屏幕的 `ECHO`。第二行让屏幕回显从输入得到的任何新行。

到目前为止，程序所做的所有工作都是改变 `termios` 结构中的属性值；也就是说，程序操纵内存中的数据。实际上下一步才更新终端。要加入修改信息，使用 `tcsetattr` 函数，如第三段代码所示。`TCSAFLUSH` 标志丢弃了用户键入的任何输入，以保证稳定一致地读操作。但是因为不是所有的终端都支持所有的 `termios` 设置，所以 POSIX 标准允许 `termios` 隐式地忽略掉它所不支持的终端功能。结果，健壮的程序应该检查以确保 `tcsetattr` 调用成功，然后在第四段确保关闭 `ECHO` 并打开 `ECHONL`。如果更新失败，则添加适当的代码来处理这种错误。

随着这些必要的设定工作结束，程序接着提示输入密码，在检索密码后打印输出用户输入的信息。在输入时不向屏幕回显字符，但对 `fprintf` 的调用则按预期的情况执行。正如刚才说明的那样，在退出程序之前的最后一步是恢复终端原来的 `termios` 状态。`err_quit` 函数显示诊断消息并在退出前恢复原来的终端属性。下面显示了这个程序的输出可能的样子：

```

$ ./noecho
Enter password:
You typed: foobar
$

```

如果你键入的口令字是8个字符或者更长，命令提示符会出现在“You typed:”一行的末尾，因为在这个例子里，口令字被限制到8个字符长。

22.4 改变终端模式

程序清单 22.1 中的程序控制了一些终端属性，但是它仍然使用规范模式。程序清单 22.2 中的程序使终端进入原始模式，并对特殊字符和信号进行自己的处理。

程序清单 22.2 setraw.c

```
/*
 * setraw.c - Illustrate using raw mode
 */

#include <termios.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

void err_quit(char *msg);
void err_reset(char *msg, struct termios *flags);
static void sig_caught(int signum);

int main(void)
{
    struct termios new_flags, old_flags;
    int i, fd;
    char c;

    /* Set up a signal handler */
    if(signal(SIGINT, sig_caught) == SIG_ERR) {
        err_quit("Failed to set up SIGINT handler");
    }
    if(signal(SIGQUIT, sig_caught) == SIG_ERR) {
        err_quit("Failed to set up SIGQUIT handler");
    }
    if(signal(SIGTERM, sig_caught) == SIG_ERR) {
        err_quit("Failed to set up SIGTERM handler");
    }

    fd = fileno(stdin);

    /* Set up raw/non-canonical mode */
    tcgetattr(fd, &old_flags);
    new_flags = old_flags;
    new_flags.c_lflag &= ~(ECHO | ICANON | ISIG);
    new_flags.c_iflag &= ~(BRKINT | ICRNL);
```

```

new_flags.c_oflag &= -OPOST;
new_flags.c_cc[VTIME] = 0;
new_flags.c_cc[VMIN] = 1;
if(tcsetattr(fd, TCSAFLUSH, &new_flags) < 0)
    err_reset("Failed to change attributes", &old_flags);

/* Process keystrokes until DELETE key is pressed */
puts("In RAW mode. Press DELETE key to exit");
while((i = read(fd, &c, 1)) == 1) {
    if((c &= 255) == 0177) {
        break;
    }
    printf( "%o\n", c);
}

/* Restore original terminal attributes */
tcsetattr(fd, TCSANOW, &old_flags);

exit(EXIT_SUCCESS);
}

void sig_caught(int signum)
{
    printf ("signal caught: Wn", signum);
}

void err_quit(char *msg)
{
    fprintf(stderr, "%s\n", msg);
    exit(EXIT_FAILURE);
}

void err_reset(char *msg, struct termios *flags)
{
    fprintf(stderr, "%s\n", msg);
    tcsetattr(fileno(stdin), TCSANOW, flags);
    exit(EXIT_FAILURE);
}

```

`err_quit` 和 `err_reset` 这两个出错处理函数的存在缩短了代码的长度。`setraw` 还创建了一个信号处理函数 `sig_caught`，来演示在原始模式下怎样捕获信号。程序真正的核心出现在 `tcgetattr` 和 `tcsetattr` 调用之间。首先，`setraw` 关闭规范模式、本地回显以及忽略信号。接下来，它关闭“回车转新行 (CR-to-NL)”的转换功能，然后关闭对输出的所有处理。在更新终端属性后，`while` 循环从输入一次读取一个字符，并把它以八进制形式显示在屏幕上。如果你按下了 Delete 键，程序在恢复原来的终端设置之后退出。

为了产生图 22.3 所示的输出，依次键入：“L” “I” “N” “U” “X” “Ctrl+C” “Ctrl+D” “Ctrl+Z” “Delete”。

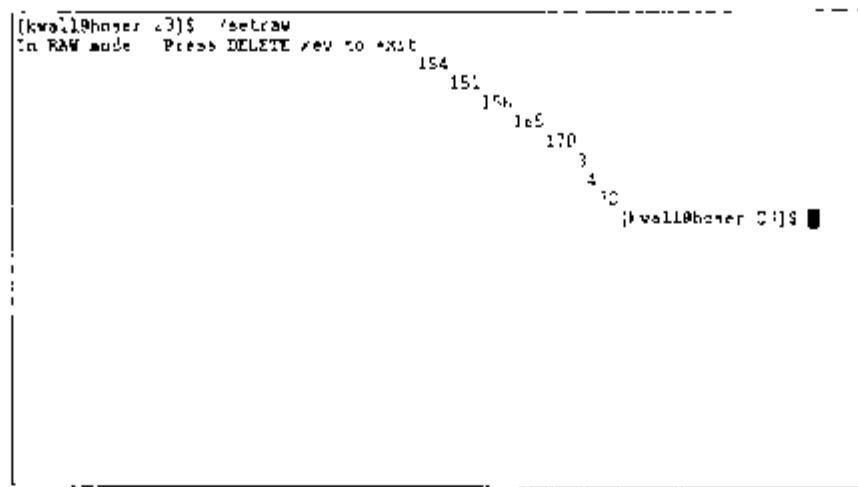


图 22.3 原始模式下的输入处理

因为“新行转回车 (newline-to-carriage return)”的转换功能被禁用，所以输入在一个新行上显示，但是在上一行的光标位置。因为程序关闭了所有的信号，所以 `setraw` 忽略了在键盘上输入的信号：`^C`（中断）、`^D`（EOF）和`^Z`（挂起）。

22.5 使用 terminfo

`termios` 提供了一个非常底层的对输入处理的控制，而 `terminfo` 则提供了一个相似层次的对输出处理的控制。`terminfo` 提供了一个可移植的低层的操作接口，比如清除终端屏幕，定位光标或是删除行或字符。由于现存终端的多样性，UNIX 的设计者们（终于）学会开始在集中存储的数据库文件中进行终端属性（能力）描述的标准化。“`terminfo`”一词的意义既指这个数据库，又指用来访问数据库的例程。

注意： 严格来说，`termcap` 数据库，首先出现在由 BSD 衍生的系统中，出现在 `terminfo` 之前（“`termcap`”的名字是“TERMinal CAPability”即终端能力一词的缩写）。然而，随着 `termcap` 数据库越来越庞大，对交互应用来说它变得非常慢，因此从第二版（SVR2）开始逐渐被从 System V 衍生的 UNIX 系统中的 `terminfo` 取代。

22.5.1 terminfo 能力

对每一种可能的终端类型，例如 VT100 或 `xterm`，`terminfo` 维护了其终端能力和特性的一个列表，被称为 `capname`，或者 CAPability NAME。`Capname` 分为下面的几类：

- 布尔型
- 数值型
- 字符串型

布尔型的 `capname` 只是简单的表明一个终端是否支持某一特定属性，比如光标寻址或快速清屏功能。数值型的 `capname` 通常定义和大小有关的能力：例如，一个终端的行数和列数分别是多少。字符串型的 `capname` 定义访问某种特性的转义序列（escape sequence），

或者定义当用户按下任意键，比如一个功能键时的输出字符串。表 22.2 列出了一小部分 terminfo 知道的能力；如果想查找完整列表请参阅用户手册页 terminfo(5)。

表 22.2 常用的终端能力

capname	类型	描述
am	布尔型	终端具有自动页边空白
bce	布尔型	终端使用背景色清除
km	布尔型	终端有 META 键
ul	布尔型	下划线字符加粗
cols	数值型	当前终端的列数
lines	数值型	当前终端的行数
colors	数值型	当前终端支持的颜色数
clear	字符串型	清屏并将光标移到初始位置
cl	字符串型	清除一直到行尾的内容
ed	字符串型	清除一直到屏幕末尾的内容
smcup	字符串型	进入光标寻址模式
cup	字符串型	将光标移至（行，列）
rmcup	字符串型	退出光标寻址模式
bel	字符串型	发出蜂鸣响声
flash	字符串型	发出可视的蜂鸣（闪烁屏幕）
kf[0-63]	字符串型	功能键 F[0-63]

为了使用 terminfo，需要在源文件中按下面的顺序包含 `<curses.h>` 和 `<term.h>`。你还必须链接 curses 函数库，因此需要在调用编译程序时加上 `-lcurses`。

22.5.2 terminfo 编程

下面是伪代码中常用的 terminfo 指令序列：

1. 初始化 terminfo 数据结构。
2. 检索 capname。
3. 修改 capname。
4. 将经过修改的 capname 输出到终端上。
5. 其他必需的代码。
6. 重复第 2~5 步。

初始化 terminfo 的数据结构很简单，如下面的代码所示：

```
#include <curses.h>

#include <term.h>

int setupterm(const char *term, int filedес, int *errret);
```

term 指定终端类型。如果 term 为 null，setupterm 读取环境变量 \$TERM 值；否则，

`setupterm` 使用 `term` 所指的终端类型值。所有的输出都被发送到由文件描述符 `filedes` 指示的文件或设备。如果 `errret` 不是 `null`，它或者是 1，这表示调用成功；或者是 0，如果在 `terminfo` 数据库中没有发现 `term` 所指示的终端；或者是 -1，如果 `terminfo` 数据库没有被找到。`Setupterm` 返回 OK 表示调用成功；返回 ERR 表示失败。如果 `errret` 是 `null`，`setupterm` 发出一个诊断信息，然后退出。

这三类能力中的每一种（布尔型、数值型和字符串型）都有一个相应的函数来检索这些能力值，它们的描述如下所示：

```
int tigetflag(const char *capname);
```

如果在 `setupterm` 中由 `term` 指定的终端支持 `capname`，则返回 TRUE；如果不支持，则返回 FALSE；如果 `capname` 不是布尔型的能力，则返回 -1。

```
int tigetnum(const char *capname);
```

返回 `capname` 的数值；如果在 `setupterm` 中由 `term` 指定的终端不支持 `capname`，返回 ERR；如果 `capname` 不是数值型的能力，则返回 -2。

```
char *tigetstr(const char *capname);
```

返回一个指向 `char` 的指针，该字符包含 `capname` 的转义序列。如果在 `setupterm` 中由 `term` 指定的终端不支持 `capname`，则返回 `(char *)null`；如果 `capname` 不是字符串型的能力，则返回 `(char *)-1`。

程序清单 22.3 使用了这些函数来查询和打印当前终端的一些能力。

程序清单 22.3 getcaps.c

```
/*
 * getcaps.c - Get and show terminal capabilities using
 *             terminfo data structures and functions
 */
#include <stdlib.h>
#include <stdio.h>
#include <term.h>
#include <curses.h>

#define NUMCAPS 3

int main(void)
{
    int i;
    int retval = 0;
    char *buf;
    char *boolcaps[NUMCAPS] = { "am", "bee", "km" };
    char *numcaps[NUMCAPS] = { "cols", "lines", "colors" };
    char *strcaps[NUMCAPS] = { "cup", "flash", "hpa" };

    if(setupterm(NULL, fileno(stdin), NULL) != OK) {
        perror("setupterm()");
        exit(EXIT_FAILURE);
    }
}
```

```

    }

    for(i = 0; i < NUMCAPS; ++i) {
        retval = tigetflag(boolcaps[i]);
        if(retval == FALSE)
            printf("' %s' unsupported\n", boolcaps[i]);
        else
            printf("' %s' supported\n", boolcaps[i]);
    }
    fputc('\n', stdout);

    for(i = 0; i < NUMCAPS; ++i) {
        retval = tigetnum(numcaps[i]);
        if(retval == ERR)
            printf("' %s' unsupported\n", numcaps[i]);
        else
            printf("' %s' is %d\n", numcaps[i], retval);
    }
    fputc('\n', stdout);

    for(i = 0; i < NUMCAPS; ++i) {
        buf = tigetstr(strcaps[i]);
        if(buf == NULL)
            printf("' %s' unsupported\n", strcaps[i]);
        else
            printf("' %s' is \\E%s\n", strcaps[i], &buf[1]);
            /*printf("' %s' is %s\n", strcaps[i], buf);*/
    }

    exit(EXIT_SUCCESS);
}

```

程序从初始化 terminfo 数据结构并为内部使用设置一些变量开始。在调用 `setupterm` 之后，它就查询并打印输出三类终端功能每一类的三个属性值。首先，在第一个 `for` 循环中检测对 `am`（自动页边空白）、`bce`（背景色清除）以及 `km`（META 键）布尔型能力的支持。然后，在第二个 `for` 循环中查询数值型能力 `cols`（终端具有的列数）、`lines`（终端能显示的行数）和 `colors`（终端显示的颜色数）。第三个 `for` 循环尝试检索三个字符串能力，`cup`（定位光标的转移序列）、`flash`（生成可视化蜂鸣的转移序列）和 `hpa`（绝对水平光标定位）。

惟一复杂的部分是第 51~52 行。如果终端支持一个特定的字符串型能力，`tigetstr` 将返回一个指向包含调用该能力所必须的转义序列的字符指针。如果你使用 `printf` 或 `puts` 将那个字符串输出到终端上，大多数情况下你实际上是调用了那个转义序列。因此，为了避免这种调用，比如可视蜂鸣（闪烁屏幕），程序去除了第一个字符，`E`（ESC），并打印出剩下的字符串。然而，出于我们的目的，这种方法并不是最优的，因为它并没有打印全部的转义序列。因此，`getcaps` 使用 `\\E` 打印静态文本，来避免转义的发生。

为了看一看如果我们没有做这种预处理，`getcaps` 将如何工作，在编译和运行程序前，注释掉语句：

```
printf("' %s' is \\E%s\n", strcaps[i], &buf[1]);
```


并取消以下代码的注释：

```
/*printf("'s' is %s\n", strcaps[i], buf);*/
```

结果将显示如下：

```
/*printf("'s' is \\E%s\n", strcaps[i], &buf[1]);*/  
printf("'s' is %s\n", strcaps[i], buf);
```

如果你的终端支持可视蜂鸣（一般的终端都支持），屏幕将发生闪烁。

在各种终端模拟器中运行 `getcaps` 显示终端功能的变化。第一个例子给出了在一个 KDE `konsole` 窗口中运行的 `getcaps`：

```
$ ./getcaps  
'am' supported  
'bce' unsupported  
'km' supported  
  
'cols' is 80  
'lines' is 24  
'colors' is 8  
  
'cup' is \E[%i%p1%d;%p2%dH  
'flash' unsupported  
'hpa' unsupported  
$
```

使用 `xterm-color` 模拟器，输出为：

```
$ ./getcaps  
'am' supported  
'bce' supported  
'km' supported  
  
'cols' is 80  
'lines' is 24  
'colors' is 16  
  
'cup' is \E[%i%p1%d;%p2%dH  
'flash' is \E[?5h  
'hpa' is \E[%i%p1%dG  
$
```

从这些例子可以得出的结论是终端的能力变化范围很大，你的代码应该考虑到这一点，在调用一种特殊功能前，首先查询 `termcap` 数据库。

注意： 命令 `infocmp` 可以列出一个终端类型的所有能力。为了看 `infocmp` 的运行，请输入命令 `infocmp -v $TERM`。为了获取更多的信息，请参阅用户手册页 `infocmp(1)`。

22.5.3 发挥 terminfo 能力

因为你已经知道如何获得终端的能力，下一步就要更新它们并使它们生效。`tparm` 用于修改 `capname`；`putp` 和 `tputs` 将变化输出到屏幕上。

函数 `putp` 假设输出设备是标准输出（`stdout`）。结果它的调用形式非常简单。其函数原型定义如下所示：

```
int putp(const char *str);
```

`putp` 将 `str` 输出到标准输出上。它等价于 `tputs(str, 1, putchar)`。

而另一方面，`tputs` 给了你很大程度的控制权，因此具有一个相应的复杂得多的接口。它的原型定义如下：

```
int tputs(const char *str, int affcnt, int (*putc)(int));
```

`tputs` 将 `str` 输出到 `setupterm` 中指定的 `term` 和 `filedes`。`str` 必须是一个 `terminfo` 字符串变量或调用 `tparm` 或 `tigetstr` 的返回值。如果 `str` 是与行相关的能力，则 `affcnt` 是受影响的行数；否则返回 1。`putc` 是一个指向任何 `putchar` 风格输出函数（一次仅输出一个字符）的指针。

一般的，在你能发送一个控制字符串之前，你必须构造它。这就是 `tparm` 函数的工作。它的定义如下：

```
char *tparm(const char *str, long p1, long p2, ..., long p9);
```

`tparm` 使用 `str` 和参数 `p1~p9` 构造了一个被恰当格式化了的参数化字符串型 `capname`。它在成功调用时返回一个指针，该指针指向一个经过更新且包含新参数 `p1~p9` 的 `str` 拷贝；而在调用失败时返回 `NULL`。

那么什么是参数化的字符串呢？让我们回忆一下 `tigetstr` 函数，它返回用来调用一个终端字符串型能力的转义序列。当你执行程序清单 22.3 的时候，应该有一个被打印到标准输出的行类似于下面的样子：

```
EcupE is \E[%i%p1%d;%p2%dH
```

这就是一个参数化的字符串。它之所以被称为参数化的字符串是因为它定义了一个一般化的能力；在这个例子中，将光标移动到屏幕上一个特定位置就是一个参数化的字符串，因为该能力还需要参数值。在 `cup` 能力的情况下，它需要行数和列数才能知道光标应放在什么位置。要在一个标准（单色）`xterm` 上移动光标，`xterm` 期望这样的命令，用接近可理解的语言表示，是下面的样子：

```
Escape -[-<row>-;-<column> -H.
```

而使用“`terminfo`”式的语言，这将是 `\E [%i%p1%d;%p2%dH`。程序员的工作就是提供行数和列数，它们由 `%pN` 所指示，这里 `N` 是一个 1~9 之间的整数值。于是，要移动光标到屏幕的 (10,15)，就有 `p1=10`，`p2=15`。为清楚起见，表 22.3 中描述了上面的 `terminfo` 字符和参数。

表 22.3 terminfo 字符和参数

字符或参数	描述
\E	将 Escape 输出到屏幕
[将 “[” 输出到屏幕
%i	将所有的%p 值加 1
%p1	将%p1 中的值压入一个内部 terminfo 堆栈
%d	将%p1 中的值弹出堆栈并输出到屏幕上
;	将 “;” 输出到屏幕上
%p2	将%p2 中的值压入（同）一个内部 terminfo 堆栈
%d	将%p2 中的值弹出堆栈并输出到屏幕上
H	将 “H” 输出到屏幕上

上面看起来好像很复杂而含糊，但考虑一下另外一个选择：为了考虑到所有你的程序可能要用到的终端，你不得不在程序中加入数百行与终端相关的需要硬编程的代码。参数化字符串建立了一个用来在所有终端类型上取得同样效果的一般格式；程序员所要做的只是用正确的值来替代这些参数。依我看来，用一些看起来好像有些神秘的字符串来替代数百行的代码实在是划算！

为了使所有这些更加具体，请看程序清单 22.4。它对上一个程序清单 22.3 进行了重写，使用了其他的终端能力来创建一个看起来更加赏心悦目的屏幕。

程序清单 22.4 new_getcaps.c

```

/*
 * new_getcaps.c - Get and show terminal capabilities using
 *                  terminfo data structures and functions
 */
#include <stdlib.h>
#include <stdio.h>
#include <term.h>
#include <curses.h>
#include <unistd.h> /* for sleep() */

#define NUMCAPS 3

void clrscr(void);
void mv_cursor(int, int);

int main(void)
{
    char *boolcaps[NUMCAPS] = { "am", "bce", "km" };
    char *numcaps[NUMCAPS] = { "cols", "lines", "colors" };
    char *strcaps[NUMCAPS] = { "cup", "flash", "hpa" };
    char *buf;
    int retval, i;

    if(setupterm(NULL, fileno(stdout), NULL) != OK) {

```

```
        perror("setupterm()");
        exit(EXIT_FAILURE);
    }

    clrscr();
    for(i = 0; i < NUMCAPS; ++i) {
        /* position the cursor */
        mv_cursor(i, 10);
        retval = tigetflag(boolcaps[i]);
        if(retval == FALSE)
            printf("'s' unsupported\n", boolcaps[i]);
        else
            printf("'s' supported\n", boolcaps[i]);
    }
    sleep(3);

    clrscr();
    for(i = 0; i < NUMCAPS; ++i) {
        mv_cursor(i, 10);
        retval = tigetnum(numcaps[i]);
        if(retval == ERR)
            printf("'s' unsupported\n", numcaps[i]);
        else
            printf("'s' is %d\n", numcaps[i], retval);
    }
    sleep(3);

    clrscr();
    for(i = 0; i < NUMCAPS; ++i) {
        mv_cursor(i, 10);
        buf = tigetstr(strcaps[i]);
        if(buf == NULL)
            printf("'s' unsupported\n", strcaps[i]);
        else
            printf("'s' is \\E%s\n", strcaps[i], &buf[1]);
    }
    sleep(3);

    exit(0);
}

/*
 * Clear the screen
 */
void clrscr(void)
{
    char *buf = tigetstr("clear");
    putp(buf);
}
```

```
/*
 * Move the cursor to the specified row row and column col
 */
void mv_cursor(int row, int col)
{
    char *cap = tigetstr("cup");
    putp(tparm(cap, row, col));
}
```

这两个程序之间的变化非常小。新加入的部分包括 `clrsrc` 和 `mv_cursor` 函数，它们分别用来清屏和把光标移至一个特定位置。它们在关闭 `main` 函数的花括号之后定义，并且利用了 `putp` 函数，因为它们更新了标准输出。

处理工具函数之外，`new_getcaps` 在 `main` 函数中添加了 9 行代码。在进入能够检索和显示终端能力的代码段之前，程序执行了清屏操作。在每段代码内，新增的调用 `mv_cursor` 分别把屏幕上的光标定位在第一、第二和第三行的第 10 列。最后，在退出每个 `for` 循环之前，`sleep` 语句使你可以查看输出的结果并看到 `terminfo` 函数调用的工作。

在结束本章之前，还有最后一点要说明。我发现这些用来演示 `termios` 和 `terminfo` 特性的程序既没有进行优化，效率也不高。我避免使用快速而高效的代码是为了教学的清晰性。例如，程序清单 22.3 和程序清单 22.4 中的三个 `for` 循环可以用一个函数指针压缩为单独一个循环。我宁愿让演示清晰明确而不愿让你费神来分析 C 语言的语法结构。

22.6 小 结

本章详细地介绍了两个用于操纵终端的底层接口。然而，这是一个很大的题目，因此这里我们并没有讨论所有的内容。但是你已经看到，虽然 `termios` 和 `terminfo` 都是对广泛多样的输入输出硬件的一般总结，可它们还是代码密集型 API。关于其他资料，请参考 `terminfo(5)`、`termios(3)` 和 `term(5)` 的手册页面。权威的技术参考资料是 O'Reilly 公司出版的《`termcap & terminfo`》一书，作者是 John Strang、Linda Mui 和 Tim O'Reilly。下一章会看到 `ncurses`，一种使用函数库来操纵终端的更容易的方式。

第 23 章 ncurses 入门

本章介绍 ncurses，它是经典的 UNIX 屏幕控制库 curses 的免费实现。ncurses 提供了用于屏幕控制和操作的一种简单的高层接口。除了拥有一系列丰富的屏幕外观控制函数之外，ncurses 还提供了处理键盘和鼠标输入、创建和管理多个窗口、使用窗体和面板的强大例程。

23.1 ncurses 简史

ncurses 是“new curses (新版 curses)”的缩写，它是随贝尔实验室的 System V Release 4.0 (SVR4) UNIX 一同发布的 curses 库的自由发布克隆版本。curses 这个词起源于短语“cursor optimization (光标优化)”，它简单地描述了光标的移动行为。接下来，SVR4 的 curses 软件包又是 System II UNIX 提供的 curses 库的一个继续演变版本，而 System II UNIX 的 curses 库本身又基于随 BSD (Berkeley Software Distribution) 版 UNIX 发行的原始 curses 库的实现。

curses 是怎样产生的呢？正如你在前一章中看到的那样，使用 termios——或者更糟的是使用 tty 接口——控制屏幕的外观显示需要编写大量的代码。另外，它同样是终端相关的，要受到大量现有的终端类型和终端模拟器特殊特性的影响。让我们来看一个古老的基于文本的冒险类游戏 rogue。在 Berkeley，Ken Arnold 将 rogue 的基于 termcap 的屏幕处理和光标移动例程汇集到一个函数库中。这个函数库最先是随 BSD UNIX 发布的。AT&T（也就是人们熟知的贝尔实验室）的 System III UNIX 包含了一个大幅度改进的 curses 函数库和 terminfo 终端描述数据库，两者都是由 Mark Horton 开发编写的。Horton 的 curses 库实现包含了对彩色显示终端和其他视频属性的支持。

System V UNIX 继续对 curses 库的特性进行扩展，增加了对窗体、菜单和面板的支持。窗体使程序员可以创建易用的数据项并能显示窗口，这简化了通常不但困难而且与应用相关的编码任务。面板扩展了 curses 库处理重叠和堆迭窗口的能力。菜单则提供了一个简单而普通的接口。

ncurses 是从 Pavel Curtis 的 pcurses 程序包直接演变而来的。Zeyd Ben-Halim 在 Curtis 工作的基础上开发了 ncurses。Eric Raymond^①又集成了许多增强功能，并继续进行 ncurses 的开发。Juergen Pfeifer 又向 ncurses 程序包加入了大部分的窗体和菜单支持。目前，ncurses 由 Thomas Dickey 来维护。他完成了 ncurses 程序包在多种系统上的配置工作，并进行了其中绝大部分的测试工作。这些年来，数以千计的程序员对该程序包贡献出了他们的改进程序和补丁程序（这一数目远远超过了目前随源程序发布的 NEWS 文件里所列举的那些人，文件中仅仅列举了从 1.9.9e 版开始直至现在的贡献者们）。

① 译者注：Eric Raymond 最著名的还是他的著作“大教堂与集市”。

ncurses 的当前版本为 4.2，而 5.0 版目前正在进行 beta 测试。关于 ncurses 有这样一个有趣的讽刺故事，意思是说现在 ncurses 已被认可为 4.4BSD “经典” curses 库的替代者，从而在绕了一个大圈子后又回到了最初产生它的操作系统上。

23.2 使用 ncurses 编译程序

在开始之前，要确认你的系统中已经安装好了开发库和包含文件。许多系统都只提供共享库，因此应用程序是在运行时加载使用 curses 的。为了使用 ncurses 编译一个程序，你需要它的函数和变量定义，所以应该在你的源代码中包含头文件<curses.h>：

```
#include <curses.h>
```

许多 Linux 系统将/usr/include/curses.h 作为头文件/usr/include/ncurses.h 的一个符号链接，因此只要包含头文件< ncurses.h > 即可。然而，为了获得最大的可移植性，应尽量使用< curses.h >，因为不管你信还是不信，ncurses 并不是在所有的 UNIX 或者类 UNIX 平台下都存在的。同时，你还需要链接 ncurses 库，因此在链接目标程序时，要使用-lcurses 选项，或者在 LDFLAGS make 变量或环境变量\$LDFLAGS 中加入-lcurses：

```
$ gcc curses_prog.c -o curses_prog -lcurses
```

23.3 调试 ncurses 程序

默认情况下，在 ncurses 程序中，调试跟踪选项是关闭的。为了启动调试功能，应链接 ncurses 库的调试库 ncurses_g，并且在你的代码中或者调用 trace(N)，或者将环境变量 \$NCURSRS_TRACE 设置为 N，其中 N 是一个正的非零整数。这样做强制将调试输出到一个名为 trace 的文件中，该文件保存在当前目录下。N 的取值越大，调试输出的结果粒度越细。N 的有效取值记录在文件< ncurses.h >中。例如，标准的跟踪级别，TRACE_ORDINARY，是 31。

提示： ncurses 程序包带有一个脚本文件 tracemunch，它用来将调试信息压缩和汇总成一种更加可读的，用户界面友好的格式。

23.4 关于窗口

本节讨论 ncurses 中关于窗口、屏幕和终端的概念。其中有几个名词将在本章中反复使用（值得庆幸的是，这些名词的使用是一致的），因此在下面的列表中我们预先给出这些名词的定义，以尽可能地避免混淆。

屏幕 (screen) 指在字符或控制台模式下的物理终端显示屏。在 X Window 系统下, “screen” 表示一个终端模拟窗口。

窗口 (window) 用来描述屏幕上显示的一个独立的矩形区域。窗口可以和屏幕大小一样, 也可以不一样。

ncurses 应用总是定义一个 stdscr, stdscr 是一个 ncurses 的数据结构, 一个指向 WINDOW 结构的指针 (WINDOW *), 用来表示当前你在屏幕上所看到的。它既可以是一个或者一组窗口, 但它一定填满了整个屏幕。你可以把它想象为一块可以在其上使用 ncurses 函数绘制的调色板。

curscr 是另一个预定义的 ncurses 变量, 也是另外一个指向 WINDOW 数据结构的指针。curscr 包含了 ncurses 关于当前屏幕是什么样子的概念。像 stdscr 一样, 它的大小也和屏幕的大小相同。curscr 和 stdscr 之间的区别在于屏幕上所出现的变化。

刷新 (refresh) 既用来指一个 ncurses 的函数调用, 又用来指一个逻辑过程。函数 refresh 将 curscr 和 stdscr 进行比较, 更新对 curscr 的任何改变, 然后将这些变化显示到屏幕上。刷新同时也被用来表示更新屏幕的整个过程。

光标 (cursor) 像刷新一样, 有两个类似的意思, 但总是指下一个字符将被显示的位置。在一个屏幕上 (物理屏幕), 光标指示着物理光标的位置。在一个窗口中 (ncurses 窗口), 它指下一个字符将被显示的逻辑位置。在本章中, 我们一般用它的第二种意思。ncurses 库用一个有序数对 (y,x) 在窗口中对光标进行定位。

23.4.1 ncurses 窗口设计

ncurses 按合理的、常规的方式定义窗口的布局。在 ncurses 中, 窗口是这样被安排的: 左上角坐标是 (0,0), 右下角坐标是 (LINES, COLS), 如图 23.1 中所示 (COLS = COLUMNS)。

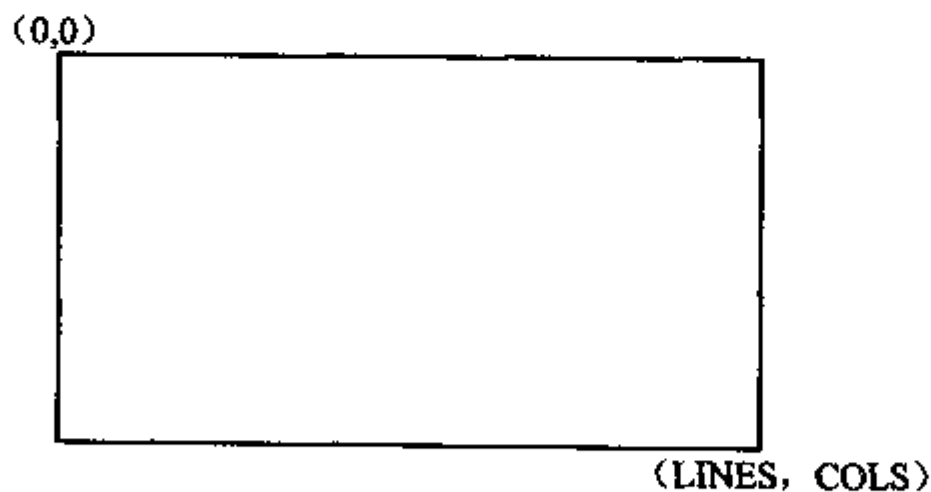


图 23.1 一个 ncurses 窗口

你可能对环境变量 \$LINES、\$COLS 已经很熟悉了。这些变量在 ncurses 库中具有对等变量, 那就是 LINES 和 COLS, 它们分别包含 ncurses 关于当前窗口大小的行数和列数的定义。然而, 我们一般不使用这些环境变量, 而使用函数调用 getmaxyx 来获得当前工作窗口的大小。

除了完全地摆脱了依赖于终端的代码以外, ncurses 的一个最主要的优点是在 ncurses 提供的 stdscr 之外创建和管理多个窗口的能力。这些由程序员定义的窗口主要有两大类, 分别是子窗口和独立窗口。

子窗口由函数调用 `subwin` 来创建。它们之所以被称为子窗口，是因为它们从一个现存的窗口来创建一个新的窗口。在 C 语言这一级，它们是指向一些指向一个现存 `WINDOW` 结构的某个子集的指针的指针。该子集能够包含整个窗口或仅仅是窗口的一部分。子窗口，也可以称为孩子窗口或派生窗口，可以对其进行独立管理，而不需要经过它的父窗口，但对于窗口所做的任何改变都应当反映到父窗口中去。

创建新的或者独立的窗口使用函数调用 `newwin`。该函数返回一个指向一个新的 `WINDOW` 结构，并与其他任何窗口都没有连接关系的指针。除非明确的要求，否则对一个独立窗口所做的改变不会在屏幕上显示出来。函数 `newwin` 对你的编程技能加入了强大的屏幕控制能力，但是正如其他附加功能的情况一样，这同样也加入了复杂性。你必须跟踪独立窗口，并且显式的将其显示到屏幕上，而子窗口则可以自动在屏幕上进行更新。

23.4.2 ncurses 函数命名规则

虽然许多 `ncurses` 的函数默认被定义为使用 `stdscr`，但在许多情况下你想要对一个不是 `stdscr` 的窗口进行操作。`ncurses` 中的例程使用一个系统的、被一致使用的命名约定，它可以被应用到任何窗口。一般而言，可以对任意窗口进行操作的函数以字符“w”作为前缀，并将一个指向 `WINDOW` 结构的指针变量 (`WINDOW *`) 作为它的第一个参数。这样的话，举个例子，函数调用 `move(y,x)`，用来将光标在 `stdscr` 上移动到由 `y` 和 `x` 指定的坐标位置，就可以被函数调用 `wmove(win,y,x)` 来代替，它用来在窗口 `win` 上将光标移动到指定的位置。这样的话，`move(y,x)` 等价于 `wmove(stdscr,y,x)`；

注意： 实际上，大多数应用于 `stdscr` 的函数都是伪函数。它们是用 `#` 定义的预处理器宏，使用 `stdscr` 作为调用与窗口相关函数的默认参数。这是一个实现的细节，你不需要过多的了解，但这可以帮助你更好地理解 `ncurses` 库。一个 `grep '#define' /usr/include/ncurses.h` 命令将很快显示出 `ncurses` 使用宏定义的程度，并可作为一个预处理器使用的很好的例子。

同样的，许多 `ncurses` 的输入、输出函数具有将一个移动操作和一个输入/输出操作结合在一个函数调用中的形式。这些函数在函数名称前加上 `mv`，将需要的 `(y,x)` 坐标加入参数表中。

因此，举个例子，你可以编写：

```
move(y,x);  
addchstr(str);
```

来在 `stdscr` 上移动光标到指定的坐标，并在那个位置加入一个字符串。或者，你可以简单地写为：

```
mvaddchstr(y,x,str);
```

从而仅用一行代码完成了同样的工作。

正如此时你可能猜想到的一样，同样存在针对某个具体窗口将输入/输出和移动操作结合在一起的函数调用。因此像下面这样的代码：

```
wmove(some_win, y, x);  
waddchstr(some_win, str);
```

可以用下面的一行代码代替：

```
mvwaddchstr(some_win, y, x, str);
```

这种速记方式充斥着整个 ncurses。该约定既简单又容易使用。

23.5 初始化和终止

在使用 ncurses 之前，必须正确地对 ncurses 子系统进行初始化，对各种 ncurses 数据结构进行设置，并查询支持终端的显示能力和特性。类似的，在推出一个基于 ncurses 的应用程序之前，你需要归还由 ncurses 分配的内存资源，并重新将终端设置为其原始的、在调用 ncurses 之前的情况。

23.5.1 ncurses 初始化结构

函数 `initscr` 和 `newterm` 处理 ncurses 的初始化要求。`initscr` 有两个任务，分别是用来创建和初始化 `stdscr` 和 `curscr`，以及通过查询 `terminfo` 和 `termcap` 数据库来发现终端的能力和特性。如果它不能完成这些任务中的某个任务，或者如果某些其他的错误发生了，`initscr` 将显示有用的检测信息，并终止应用程序的执行。在使用任何其他操纵 `stdscr` 和 `curscr` 的例程之前应首先调用 `initscr`。如果没有这样做，将导致应用程序由于段寻址错误而异常终止。然而同时也应注意，只有当你确信需要它时，才调用 `initscr`，比如在其他检查程序启动时错误的例程完成之后。最后，用来改变一个终端状态的函数调用，例如 `cbreak` 或 `noecho`，应该在 `initscr` 调用返回后才被调用。在 `initscr` 调用之后首先调用 `refresh` 将清除屏幕的内容。如果调用成功，`initscr` 返回一个指向 `stdscr` 的指针，你可以将其保存起来以在后面需要时使用。否则，该调用返回 `NULL`，终止程序的执行，并在显示设备上打印出一条有用的错误信息。

如果你的程序将发送输出到不止一个终端，或者从不止一个终端接收输入，使用 `newterm` 函数调用来代替 `initscr`。对于每一个你希望与之交互的终端设备，都调用一次 `newterm`。`newterm` 返回一个指向一个 `SCREEN` 类型的 C 语言数据结构（另一个 ncurses 定义的类型），用来在引用一个终端时使用。然而，在能向这样的一个终端发送输出或从这样的一个终端接收输入之前，你必须将它设置为当前终端。函数调用 `set_term` 用来完成这一任务。将指向你想使其成为当前终端的 `SCREEN` 结构的指针（由一个先前的 `newterm` 调用返回）作为 `set_term` 的参数。

23.5.2 ncurses 终止

初始化函数以一种 ncurses 友好的方式分配内存资源并重新设置终端状态。相应地，你需要释放被分配的内存资源，并重新设置终端状态到使用 ncurses 之前的模式。终止函数 `endwin` 和 `delscreen` 用来完成这项工作。当你完成了对一个 `SCREEN` 的工作，在让另外一

个终端成为当前终端之前应调用 `endwin`，然后在那个终端上调用 `delscreen` 来释放分配给它的 `SCREEN` 资源，因为 `endwin` 并不为由 `newterm` 创建的屏幕释放内存。然而，如果没有调用 `newterm`，并且只使用了 `curscr` 和 `stdscr` 的话，在退出应用程序之前所有需要做的只是调用 `endwin`。`endwin` 将光标移动到屏幕的左下角，并重新设置终端为非可见的、调用 `ncurses` 以前的状态。分配给 `curscr` 和 `stdscr` 的内存并不释放，因为你的程序能暂时地调用 `endwin` 来挂起 `ncurses` 的执行，完成其他处理，然后调用 `refresh`。

下面提供了对至今所讨论过的每一个函数的一个参考：

```
WINDOW *initscr(void);
```

在确定了当前终端类型后，初始化 `ncurses` 数据结构。调用成功时，返回一个指向 `stdscr` 的指针；失败时返回 `NULL`。

```
int endwin();
```

在调用 `initscr` 或者 `newterm` 之前恢复先前的 `tty` 状态。

调用成功时，返回 `OK`；失败时，返回 `ERR`，并终止应用程序的执行。

```
SCREEN *newterm(const char *type, FILE *outfd, FILE *infd);
```

类似于使用多个终端的程序的 `initscr` 调用。如果需要的话，`type` 是一个将被使用的字符串，用来代替环境变量 `$TERM`；如果为 `NULL`，`$TERM` 将被使用。`outfd` 指向一个被用来输出到终端的文件的指针，`infd` 指向一个被用来从终端获得输入的文件的指针。调用成功时，返回一个指向这个新产生终端的指针，该指针将被保存以备将来对终端进行引用时使用；失败时返回 `NULL`。

```
SCREEN *set_term(SCREEN *new);
```

设置当前终端为由 `new` 指定的终端，它必须是一个指向 `SCREEN` 结构的指针，由先前的一个 `newterm` 调用返回。所有后面的输入和输出以及其他的 `ncurses` 例程都在由 `new` 指定的终端上进行操作。调用成功时，返回一个指向原来旧终端的指针；失败时返回 `NULL`。

```
void delscreen(SCREEN *sp);
```

释放与 `sp` 相关联的内存。必须在 `endwin` 之后，为与 `sp` 相关联的终端进行调用。

23.5.3 说明 `ncurses` 初始化和终止

程序清单 23.1 和 23.2 说明了迄今为止我们所见到的 `ncurses` 例程的使用。第一个程序 `initcurs`，显示了标准地使用 `initscr` 和 `endwin` 的 `ncurses` 初始化和终止语句。而第二个程序 `newterm`，显示了 `newterm` 和 `delscreen` 调用的恰当使用。

程序清单 23.1 `initcurs.c`

```
#include <stdlib.h>
#include <unistd.h>
#include <curses.h>
#include <errno.h>
```

```
int main(void)
{
    if ((initscr()) == NULL) {
        perror("initscr");
        exit(EXIT_FAILURE);
    }

    printw("This is a curses window\n");
    refresh();
    sleep(3);

    printw("Going bye-bye now\n");
    refresh();
    sleep(3);
    endwin();

    exit(0);
}
```

为了必要的函数声明和变量定义，我们在第 7 行包含了 `<curses.h>` 文件。在没有任何其他起始代码的情况下，我们立即在第 12 行调用 `initscr` 对 `curses` 进行初始化（通常情况下，你需要获得它返回的 `WINDOW *`）。在第 17 行和第 21 行，我们使用函数 `printw`（该函数将在下一节进行详细地阐述）将一些输出显示到窗口中，在第 18 行和第 22 行刷新显示设备，从而使实际输出出现在屏幕上。在一个 3 秒钟的停顿（在第 19 行和第 23 行）之后，我们调用 `endwin`（第 24 行）终止程序，并且释放由 `initscr` 分配的资源。

程序清单 23.2 newterm.c

```
/*
 * Listing 23.2
 * newterm.c -curses initialization and termination
 */
#include <stdlib.h>
#include <unistd.h>
#include <curses.h>
#include <errno.h>

int main(void)
{
    SCREEN *scr;

    if ((scr = newterm(NULL, stdout, stdin)) == NULL) {
        perror("newterm");
        exit(EXIT_FAILURE);
    }

    if (set_term(scr) == NULL) {
        perror("set_term");
        endwin();
        delscreen(scr);
    }
}
```

```
        exit(EXIT_FAILURE);
    }

    printw("This curses window created with newterm()\n");
    refresh();
    sleep(3);

    printw("Going bye-bye now\n");
    refresh();
    sleep(3);
    endwin();
    delscreen(scr);

    exit(0);
}
```

该程序非常类似于第一个程序。我们使用 `newterm` 调用来初始化 `curses` 子系统（第 13 行），并假定我们将交互不同的终端。因为想让输入和输出分别置于它们通常的位置，我们传递 `stdout`（标准输出）和 `stdin`（标准输入）分别作为输出和输入的 `FILE` 指针。然而，在能够使用 `scr` 以前，我们必须将其设置为当前终端，因此有第 18 行对 `set_term` 的调用。如果该调用失败了，我们必须确保调用 `endwin` 和 `delscreen` 来释放与 `scr` 联系的内存，因此第 20 行和第 21 行，我们加入了在错误检测中完成该任务的代码。在将输出发送到我们的“终端”以后（第 25 行和第 29 行），我们使用必须的 `delscreen` 函数调用关闭 `curses` 子系统。

23.6 输入和输出

`ncurses` 包含很多用于将输出发送到屏幕和窗口以及从屏幕和窗口接收输入的函数。理解 C 的标准输入和输出例程在 `ncurses` 的窗口中不能工作这一点是很重要的。但幸运的是，`ncurses` 的输入/输出例程操作非常类似于标准的输入/输出（`<stdio.h>`）例程，因此学习起来很容易。

23.6.1 输出例程

为了讨论的方便，我们将 `ncurses` 的输出例程划分为字符、字符串和各种其他的类别。下面将详细讨论这些类别。

字符例程

`ncurses` 的核心字符输出函数是 `addch`，在 `<ncurses.h>` 中的原型定义为

```
int addch(chtype ch);
```

它将字符 `ch` 显示在当前窗口（通常是 `stdscr`）中光标的当前位置，并将光标移动到它的下一个位置。如果这一操作使光标的放置超过了屏幕的右边界，光标将自动地换行到下一行的开始位置。如果当前窗口的可滚动选项打开（使用 `scrollok` 调用），并且光标处于可

滚动区域的底部，该区域将向前滚动一行。如果 `ch` 是一个制表符、换行或者回车符，光标将进行相应的移动。其他的控制字符使用 `^X` 表示法进行显示，这里 `X` 是一个字符，而脱字符号 (^) 表示该字符是一个控制字符。如果你需要发送字面意义的控制字符 (literal control character)，使用函数 `echochar (chtype ch)`。几乎所有其他的输出函数都通过调用 `addch` 来完成他们的工作。

注意： `ncurses` 文档将类型为 `chtype` 的字符描述为“伪字符”。`ncurses` 将伪字符声明为无符号长整型，使用这些字符的高位来存储额外的信息，比如视频属性等。伪字符和普通 C 字符之间的这一区别意味着在处理每种类型的函数操作时将有细微差别。我们将在本章中的合适地方指出这些差别。

正如前面所指出的，`mvaddch` 移动光标到指定位置，并将一个字符加到指定的窗口；`mvwaddch` 将某个特定窗口上的一个移动操作和一个输出操作结合在一起。`waddch` 将一个字符显示到由用户指定的窗口中。`echochar` 函数，和另一个和它紧密联系的但可指定窗口的函数，`wechochar`，将 `addch` 调用与 `refresh` 或 `wrefresh` 调用结合在一起，从而在使用非控制字符时可使性能产生显著的提高。

使用 `chtype` 类型字符或字符串的 `ncurses` 例程的一个特别有用的特性是输出的字符或字符串可以在其显示之前与许多视频属性进行逻辑“或”操作 (ORed)。这些属性的部分列表包括：

<code>A_NORMAL</code>	标准的显示模式
<code>A_STANDOUT</code>	使用终端的最好的加亮模式
<code>A_UNDERLINE</code>	加下划线
<code>A_REVERSE</code>	使用反转视频
<code>A_BLINK</code>	闪烁文本
<code>A_DIM</code>	半强度显示
<code>A_BOLD</code>	超强度显示
<code>A_INVIS</code>	字符将成为不可见
<code>A_CHARTEXT</code>	创建一个位掩码以进行字符的提取

然而依赖于终端仿真或者屏幕硬件的能力，并不是所有的属性都是可以使用的。可参考 `curs_attr(3)` 的用户手册页以获得更多的细节。

除了控制字符和具有增强视频属性的字符以外，字符输出函数也显示行图形字符 (ASCII 字符表中后半部分的字符)，例如方块以及其他一些特殊的符号。完全的行图形符号请参考 `cur_addch(3)` 的用户手册页，下面列出了一些常见的此类字符：

<code>ACS_ULCORNER</code>	左上角
<code>ACS_LLCORNER</code>	左下角
<code>ACS_URCORNER</code>	右上角
<code>ACS_LRCORNER</code>	右下角
<code>ACS_HLINE</code>	水平线
<code>ACS_VLINE</code>	垂直线

到目前为止所描述的这些函数能够有效地向窗口添加字符，而不会影响其他已存在字符的放置。另外一组例程可以将字符插入已存在窗口文本的任意位置。这些函数包括 `insch`、`winsch`、`mvinsch` 和 `mvwinsch`。遵从本章前面讨论的命名约定，这些函数都是在光标位置的字符前面插入一个字符，将后面的字符向右移动一个位置；如果最右边的字符处于屏幕的右边界，它将丢失。然而，请注意，光标位置在一个插入操作后并不改变。插入操作在 `curs_insch(3)` 用户手册页中有详尽的叙述。目前我们所提及的函数的原型定义如下所示：

```
int addch(chtype ch);
int waddch(WINDOW *win, chtype ch);
int mvaddch(int y, int x, chtype ch);
int mvwaddch(WINDOW *win, int y, int x, chtype ch);
int echochar(chtype ch);
int wechochar(WINDOW *win, chtype ch);
int insch(chtype ch);
int winsch(WINDOW *win, chtype ch);
int mvinsch(int y, int x, chtype ch);
int mvwinsch(WINDOW *win, int y, int x, chtype ch);
```

除非另外说明，所有的函数均在成功调用后返回 `OK`，在调用失败时返回 `ERR` (`OK` 和 `ERR` 以及其他的一些常量在 `<nurses.h>` 中定义)。参数 `win`、`y`、`x` 和 `ch` 分别是字符将要在其上显示的窗口，用来定位光标的 `y` 和 `x` 坐标值，以及将要显示的字符（包括可选的属性）。最后还要提醒用户注意，以字符“w”作为前缀的例程接受一个指向目标窗口的指针 `win`；而以“mv”作为前缀的例程将一个移动到坐标(`y,x`)位置的操作和一个输出操作结合在一起。

字符串例程

`ncurses` 的字符串例程大体上与字符例程的行为类似，除了它们是处理由伪字符构成的字符串或者普通的由 `null` 终结的字符串。`ncurses` 的设计者们同样创建了一个标准的表示法来帮助程序员对这两类函数进行区分。函数名含有 `chstr` 的函数在由伪字符构成的字符串上进行操作，而函数名仅含有 `str` 的函数使用标准的 C 类型的字符串（以 `null` 作为串终结）。对伪字符串进行操作的函数的一部分列表包括：

```
int addchstr(const chtype *chstr);
int addchnstr(const chtype *chstr, int n);
int waddchstr(WINDOW *win, const chtype *chstr);
int waddchnstr(WINDOW *win, const chtype *chstr, int n);
int mvaddchstr(int y, int x, const chtype *chstr);
int mvaddchnstr(int y, int x, const chtype *chstr, int n);
int mvwaddchstr(WINDOW *win, int y, int x, const chtype *chstr);
int mvwaddchnstr(WINDOW *win, int y, int x, const chtype *chstr,
                  int n);
```

所有列出的这些函数拷贝 `chstr` 到目标窗口中，从光标的当前位置开始，但光标本身并不前移（这一点与字符输出函数不同）。如果字符串比将要放置的当前行长，它将在右边界被截断。`Addchnstr`、`waddchnstr`、`mvaddchnstr` 和 `mvwaddchnstr` 这四个例程都接收一个整数

`n` 作为参数，拷贝上界长为 `n` 个字符，并在右边界截止。如果 `n` 为 -1，整个字符串将被拷贝，但如果需要的话，将在右边界被截断。

接下来的一组字符串输出函数对以 `null` 结尾的字符串进行操作。与前面的一组函数不同，这些函数将光标前移。另外，字符串的输出将在右边界处换行，而不是被截断，它们都和具有近似命名的 `chtype` 类型的对等函数有类似的行为。

```
int addstr(const char *str);
int addnstr(const char *str, int n);
int waddstr(WINDOW *win, const char *str);
int waddnstr(WINDOW *win, const char *str, int n);
int mvaddstr(int y, int x, const char *str);
int mvaddnstr(int y, int x, const char *str, int n);
int mvwaddstr(WINDOW *win, int y, int x, const char *str);
int mvwaddnstr(WINDOW *win, int y, int x, const char *str, int n);
```

切记，这些例程中的 `str` 是一个标准的、C 类型的、以 `null` 结尾的字符数组。

接下来也是最后一组我们将看到的输出例程是一个大杂烩：用于绘制边界和线的函数调用，用于清除和设置背景的，用于控制输出选项的，用于移动光标的以及用于向一个 `ncurses` 窗口发送格式化输出数据的函数调用。

其他输出例程

为了设置窗口的背景属性，使用 `bkgd`，其原型定义为：

```
int bkgd(const chtype ch);
```

`ch` 是一个字符和前面列出的一个或多个视频属性进行“或”操作所得到的组合值。为了获得当前的背景设置，调用 `chtype getbkgd(WINDOW *win)`；这里 `win` 是我们感兴趣的窗口。设置和获得窗口背景函数的完整描述可以在 `curl_bkgd(3)` 用户手册页中获得。

至少有 11 个 `ncurses` 函数用来在 `ncurses` 窗口中绘制方块、边界和线。其中 `box` 调用最简单，它在一个指定的窗口中绘制一个矩形框，用一个字符来绘制垂直线，用另一个字符来绘制水平线。它的原型定义如下：

```
int box(WINDOW *win, chtype verch, chtype horch);
```

`verch` 设置用来绘制垂直线的伪字符，`horch` 设置用来绘制水平线的伪字符。

`border` 函数的原型如下：

```
int border(WINDOW *win, chtype ls, chtype rs, chtype ts, chtype
bs, chtype tl, chtype tr, chtype bl, chtype br);
```

参数分别为：

<code>ls</code>	左边
<code>rs</code>	右边
<code>ts</code>	上边
<code>bs</code>	下边

tl	左上角
tr	右上角
bl	左下角
br	右下角

box 和 border 函数均在窗口中沿着窗口的左、右、上和下边界绘制轮廓线。

使用 hline 函数在当前窗口上画一条任意长的水平线。同样的，使用 vline 函数画一条任意长的垂直线。

```
int hline(chtype ch, int n);
int vline(chtype ch, int n);
```

按照 ncurses 关于函数的命名约定，你可以使用下面的函数指定窗口进行画线操作：

```
int whline(WINDOW *win, chtype ch, int n);
int wvline(WINDOW *win, chtype ch, int n);
```

或者使用下面的函数将光标移动到某个特定的位置

```
int mvhline(int y, int x, chtype ch, int n);
int mvvline(int y, int x, chtype ch, int n);
```

甚至可以使用下面的函数指定窗口并进行移动操作

```
int mvwhline(WINDOW *win, int y, int x, chtype ch, int n);
int mvwvline(WINDOW *win, int y, int x, chtype ch, int n);
```

像往常一样，这些例程在调用成功时返回 OK，失败时返回 ERR。参数 win 用来指定目标窗口，n 用来指定最大的线长，其上界可以是窗口大小的垂直长或水平长。直线、框和边界的绘制函数并不改变光标的位置。接下来的输出操作能覆盖边界，因此你必须确保或者包含了维持边界完整性的函数调用，或者编写你自己的输出函数，令它们不会覆盖边界。那些不指定光标位置的函数（line、vline、whline 和 wvline）从当前的光标位置开始绘制。记录这些函数的用户手册页是 `curs_border(3)`。文档页 `curs_outopts(3)` 也含有相关的信息。

最后一组要考虑的函数用于清除全部或部分屏幕。像往常一样，它们可用于无格式和指定窗口的两种形式：

```
int erase(void);
int werase(WINDOW *win);
int clear(void);
int wclear(WINDOW *win);
int clrtobot(void);
int wclrtobot(WINDOW *win);
int clrtoeol(void);
int wclrtoeol(WINDOW *win);
```

erase 对窗口中的每一个位置写空字符；clrtobot 从当前的光标位置开始清除屏幕，直至窗口的底部（包括窗口的底部）；最后，clrtoeol 将从光标位置删除当前行，直至行的右

边界（包括右边界）。如果你已经用过 `bkgd` 或者 `wbkgd` 来设置即将被清除或删除的窗口的背景属性，那些被设置的属性将被应用到每一个创建的空字符。关于这些函数的相关用户手册页是 `curs_clear(3)`。

下面的程序展示了本节所讨论的例程有多少是可以使用的。由于这些函数调用约定极大的相似性以及如此大量的例程，这里示例程序仅仅示范了 `ncurses` 接口的使用。为了缩短程序，我们将初始化和终止代码移到一个单独的文件 `utilfcns.c` 中，并且包含头文件 `utilfcns.h`，该头文件中含有接口的定义（这两个文件都存放在随书的 CD-ROM 中）。程序清单 23.3 说明了 `ncurses` 的字符输出函数。

程序清单 23.3 `curschar.c`

```
/*
 * Listing 23.3
 * curschar.c - curses character output functions
 */
#include <stdlib.h>
#include <unistd.h>
#include <curses.h>
#include <errno.h>
#include "utilfcns.h"

int main(void)
{
    app_init();

    addch('X');
    addch('Y' | A_REVERSE);
    mvaddch(2, 1, 'Z' | A_BOLD);
    refresh();
    sleep(3);

    clear();
    waddch(stdscr, 'X');
    waddch(stdscr, 'Y' | A_REVERSE);
    mvwaddch(stdscr, 2, 1, 'Z' | A_BOLD);
    refresh();
    sleep(3);

    app_exit();
    exit(0);
}
```

正如你在第 15 行和第 16 行看到的那样，`addch` 例程输出了期望的字符并且向前移动了光标。第 16 行和第 17 行显示了如何将视频属性与要显示的字符进行结合。在第 17 行我们也显示了一个典型的“mv”为前缀的函数的使用。在刷新屏幕之后（第 18 行），一个短暂的停顿允许你观察输出结果。注意，直到执行了刷新操作以后，所做的改变才出现在屏幕上。第 22~26 行使用指定窗口的例程，并指定 `stdsrc` 作为目标窗口重复了这一过程。

程序清单 23.4 简要说明了字符串输出函数的使用。

程序清单 23.4 cursstr.c

```
/*
 * Listing 23.4
 * cursstr.c - curses string output functions
 */
#include <stdlib.h>
#include <unistd.h>
#include <curses.h>
#include <errno.h>
#include "utilfcns.h"

int main(void)
{
    int xmax, ymax;
    WINDOW *tmpwin;

    app_init();
    getmaxyx(stdscr, ymax, xmax);

    addstr("Using the *str() family\n");
    hline(ACS_HLINE, xmax);
    mvaddstr(3, 0, "This string appears in full\n");
    mvaddnstr(5, 0, "This string is truncated\n", 15);
    refresh();
    sleep(3);

    if ((tmpwin = newwin(0, 0, 0, 0)) == NULL)
        err_quit("newwin");

    mvwaddstr(tmpwin, 1, 1, "This message should appear in
        a new window");
    wborder(tmpwin, 0, 0, 0, 0, 0, 0, 0, 0);
    touchwin(tmpwin);
    wrefresh(tmpwin);
    sleep(3);

    delwin(tmpwin);
    app_exit();
    exit(0);
}
```

第 17 行调用 `getmaxyx` 检索 `stdscr` 的列数和行数信息——该例程的语法并不需要 `ymax` 和 `xmax` 作为指针类型。因为我们调用 `mvaddnstr` 时设置 `n` 的值为 15，我们想要打印的字符串将在字符串“truncated”中的字母“t”之前被截断。在第 26 行，我们创建了一个和当前窗口一样大小的新窗口 `tmpwin`。然后我们在这个新窗口中乱写一个消息（第 29 行），在它周围绘制了一个轮廓框（第 30 行），并且调用 `refresh` 将它显示在屏幕上。在程序退出以前，我们在窗口上调用 `delwin` 函数释放其资源（第 35 行）。

程序清单 23.5 显示了使用行图形字符以及 `box` 和 `wborder` 函数调用。特别要注意第 18~25 行。一些 `ncurses` 的输出例程在输出后移动光标，而其他的例程则不会移动光标。同时也要注意关于行绘图的一组函数，比如 `vline` 和 `hline`，是从上到下、从左到右绘图的，所以在使用它们时应注意光标的定位。

程序清单 23.5 `cursbox.c`

```
/*
 * Listing 23.5
 * cursbox.c - curses box drawing functions
 */
#include <stdlib.h>
#include <unistd.h>
#include <curses.h>
#include <errno.h>
#include "utilfcns.h"

int main(void)
{
    int ymax, xmax;

    app_init();
    getmaxyx(stdscr, ymax, xmax);

    mvaddch(0, 0, ACS_ULCORNER);
    hline(ACS_HLINE, xmax - 2);
    mvaddch(ymax - 1, 0, ACS_LLCORNER);
    hline(ACS_HLINE, xmax - 2);
    mvaddch(0, xmax - 1, ACS_URCORNER);
    vline(ACS_VLINE, ymax - 2);
    mvvline(1, xmax - 1, ACS_VLINE, ymax - 2);
    mvaddch(ymax - 1, xmax - 1, ACS_LRCORNER);
    mvprintw(ymax / 3 - 1, (xmax - 30) / 2, "border drawn the
        hard way");
    refresh();
    sleep(3);

    clear();
    box(stdscr, ACS_VLINE, ACS_HLINE);
    mvprintw(ymax / 3 - 1, (xmax - 30) / 2, "border drawn the
        easy way");
    refresh();
    sleep(3);

    clear();
    wborder(stdscr, ACS_VLINE | A_BOLD, ACS_VLINE | A_BOLD,
        ACS_HLINE | A_BOLD, ACS_HLINE | A_BOLD,
        ACS_ULCORNER | A_BOLD, ACS_URCORNER | A_BOLD, \
        ACS_LLCORNER | A_BOLD, ACS_LRCORNER | A_BOLD);
}
```

```
    mvprintw(ymax / 3 - 1, (xmax - 25) / 2, "border drawn with wborder")
    refresh();
    sleep(3);

    app_exit();
    exit(0);
}
```

正如你所期望的那样，第 24 行的 `mvvline` 调用在绘制一条垂直线之前先移动光标。在所有绘制一个边界的回转操作完成之后，`box` 例程将变得非常容易（第 31 行）。`wborder` 函数比 `box` 函数更加冗长，但它允许对用来绘制边界的字符进行更加精细的控制。程序展示了每一个参数的默认字符，但实际上任何字符（包括可选的视频属性）都可以，只要它可以被下层的模拟器或视频硬件所支持即可。

与 C 语言标准库函数相比，使用 `ncurses` 进行输出需要一些额外的步骤；然而值得庆幸的是，我们已经显示了这比直接使用 `termios` 或者用大量的难以阅读的换行、空格和制表符来填充自己的代码要容易的多。

23.6.2 输入例程

像它的输出例程一样，`ncurses` 的输入例程也分为几个组。然而基于两个原因，本章将仅集中于简单字符和字符串的输入。首先也是最重要的原因是，本节中的例程将满足你 90% 的需要。第二，`ncurses` 输入与 `ncurses` 输出是非常相似的，因此前面几节的材料将是一个坚实的基础。

输入的核心函数可以被压缩为三个：`getch`、`getstr` 和 `scanw`。`getch` 的原型定义如下：

```
int getch(void);
```

它从键盘获取单个字符，返回该字符或在调用失败时返回 `ERR`。它是否将获取的字符回显到 `stdscr` 上，这决定于回显属性是否被打开（这样，`wgetch` 和变量也从键盘获得单个字符，而且将它们回显还是不回显到程序指定的窗口中均可）。对于将被回显的字符，首先调用 `echo`；为了关闭回显功能，调用 `noecho`。注意在启动了回显功能以后，字符通过 `waddch` 调用显示窗口中当前的光标位置，光标也向前移动一个位置。

如果进一步考虑当前的输入模式，事情要更加复杂。输入模式用来决定在程序接收字符之前内核进行处理的程度。在一个 `ncurses` 程序中，一般你会想自己来处理绝大部分的击键事件。要这样做需要 `crmode` 模式或者 `raw` 模式（开始时 `ncurses` 处于默认模式，这意味着内核将缓存正常的文本输入，在将击键事件发送给 `ncurses` 之前等候新行的输入——但通常你不会想要这种情况）。在 `raw` 模式下，内核并不缓存或处理任何输入，而在 `crmode` 模式下，内核处理像 `^S`、`^Q`、`^C` 或 `^Y` 等终端控制字符并将所有其他的字符原封不动地传递给 `ncurses`。在某些系统中文字“下一个字符”（即 `^V`）可能需要重复。依赖于应用程序的需要，`crmode` 模式应该足够用了。在我们的例子中，就有一个实用程序使用 `crmode` 模式，启动然后关闭回显模式，用来模仿影子密码的检索。

函数 `getstr` 定义为：

```
int getstr(char *str);
```

反复地调用 `getch` 函数直到它遇到了一个换行或回车符为止（回车符本身并不是返回字符串的一部分）。输入的字符串存在 `str` 中。因为 `getstr` 函数不进行边界检查，我们强烈推荐使用 `getnstr` 函数来代替，因为后者提供了一个额外的参数用来指定存储字符的最大个数。不管你使用的是 `getstr` 还是 `getnstr` 函数，接收缓冲区 `str` 必须有足够的空间来存放接收字符串和一个表示终结的空字符，该空字符必须由程序添加。

`scanw` 函数以和 `scanf(3)` 相似的方式从键盘获得经过格式化的输入。实际上，`ncurses` 将接收的字符作为输入传递给 `sscanf(3)`，因此任何没有映射为格式域中有效参数的输入将被发送到位存储桶（一种专门存储溢出位的移位寄存器）。通常，`scanw` 具有用来进行移动操作的变量（前面加上了“mv”前缀），它可以应用于指定窗口（以“W”为前缀）。另外，`scanw` 函数族还包含了一个用于处理可变长参数表的函数成员 `vwscanw`。相关的函数原型定义如下：

```
int scanw(char *fmt [, arg] ...);
int vwscanw(WINDOW *win, char *fmt, va_list varglist);
```

用户手册 `curs_getch(3)`、`curs_getstr(3)` 和 `curs_scanw(3)` 中详细完整地记录了这些例程以及它们的各种变体函数。它们的用法显示在程序清单 23.6 和 23.7 中。

程序清单 23.6 `cursinch.c`

```
/*
 * Listing 23.6
 * cursinch.c - curses character input functions
 */
#include <stdlib.h>
#include <unistd.h>
#include <curses.h>
#include <errno.h>
#include "utilfcns.h"

int main(void)
{
    int c, i = 0;
    int xmax, ymax;
    char str[80];
    WINDOW *pwin;

    app_init();
    crmode();

    getmaxyx(stdscr, ymax, xmax);
    if ((pwin = subwin(stdscr, 3, 40, ymax / 3, (xmax - 40) / 2))
        == NULL)
        err_quit("subwin");
    box(pwin, ACS_VLINE, ACS_HLINE);
    mvwaddstr(pwin, 1, 1, "Password: ");

    noecho();
```

```

    while ((c = getch()) != '\n' && i < 80) {
        str[i++] = c;
        waddch(pwin, '*');
        wrefresh(pwin);
    }
    echo();
    str[i] = '\0';
    wrefresh(pwin);

    mvwprintw(pwin, 1, 1, "You typed: %s\n", str);
    box(pwin, ACS_VLINE, ACS_HLINE);
    wrefresh(pwin);
    sleep(3);

    delwin(pwin);
    app_exit();
    exit(0);
}

```

在成功地初始化 `ncurses` 子系统之后，我们立即切换到 `crmode` 模式（第 19 行）。创建一个带有边框的窗口将用户用来输入密码的区域和周围的区域明显区分开来（第 22~25 行）。出于对安全的考虑，我们不愿意将输入的密码显示在屏幕上，因此在读取密码的过程中，我们关闭了回显模式（第 27 行）。但是，为了给用户提供看得见的反馈，我们使用 `waddch` 函数在密码窗口中为用户键入的每一个密码字符回显一个“*”（第 30 行和第 31 行）。在遇到一个新行时，我们退出等待密码输入的循环，重新启动回显模式（第 33 行），并且给包含密码的字符串加上终结符（第 34 行），接下来将向用户显示他所键入的内容（第 37 行）。注意，`wprintw` 函数覆盖了我们绘制的包围密码窗口的矩形，因此在刷新屏幕之前我们重画了窗口的边界（第 38 行）。

程序清单 23.7 `cursgstr.c`

```

/*
 * Listing 23.7
 * cursgstr.c - curses string input functions
 */
#include <stdlib.h>
#include <unistd.h>
#include <curses.h>
#include <errno.h>
#include <string.h>
#include "utilfcns.h"

int main(int argc, char *argv[])
{
    char str[20];
    char *pstr;

    app_init();

```

```

    crmode();

    printw("File to open: ");
    refresh();
    getstr(str);
    printw("You typed: %s\n", str);
    refresh();
    sleep(3);
    if ((pstr = malloc(sizeof(char) * 20)) == NULL)
        err_quit("malloc");

    printw("Enter your name: ");
    refresh();
    getnstr(pstr, 20);
    printw("You entered: %s\n", pstr);
    refresh();
    sleep(3);

    free(pstr);
    app_exit();
    exit(0);
}

```

在程序中，我们保持回显模式一直处于打开状态，因为用户可能想看到他所输入的内容。首先我们使用 `getstr` 函数（第 22 行）。在实际的程序中，我们可能会试图打开以输入的字符串为文件名的文件。在第 32 行，我们使用 `getnstr` 函数，这样我们可以看到在输入的字符串长度超过长度界限 `n` 时，`ncurses` 是如何工作的。在这个例子中，`ncurses` 将停止对输入的接受和回显，并在你输入超过 20 个字符时发出蜂鸣警告。

23.7 色彩例程

我们已经看到，`ncurses` 支持几种不同的显示亮度模式。有趣的是，它以同样的方式提供对色彩的支持；也就是说，你可以将想要的颜色值和 `addch` 调用或其他任何接收伪字符（`chtype`）参数的输出例程的字符参数进行逻辑“或”操作。然而这种方法很啰嗦，因此 `ncurses` 还提供了一组可以对每个窗口进行显示属性设置的例程。

在使用 `ncurses` 的颜色能力之前，你必须确保当前的终端支持色彩显示。函数调用 `has_colors` 可以用来检测当前的显示设备是否支持色彩——如果支持则返回 `TRUE`；否则返回 `FALSE`。

```
bool has_colors(void);
```

`ncurses` 默认的颜色有：

<code>COLOR_BLACK</code>	黑色
<code>COLOR_RED</code>	红色

COLOR_GREEN	绿色
COLOR_YELLOW	黄色
COLOR_BLUE	蓝色
COLOR_MAGENTA	品红色
COLOR_CYAN	青色
COLOR_WHITE	白色

一旦确认了你的终端支持色彩，就可以调用 `start_color` 函数来初始化默认颜色值。

```
int start_color(void);
```

程序清单 23.8 显示了基本的颜色使用。它必须运行在一台支持颜色显示的终端模拟器上，例如彩色 X 终端（`color xterm`）。

程序清单 23.8 `color.c`

```
/*
 * Listing 23.8
 * color.c - curses color management
 */
#include <stdlib.h>
#include <unistd.h>
#include <curses.h>
#include <errno.h>
#include "utilfcns.h"

int main(void)
{
    int n;

    app_init();

    if (has_colors()) {
        if (start_color() == ERR)
            err_quit("start_color");

        /* Set up some simple color assignments */
        init_pair(COLOR_BLACK, COLOR_BLACK, COLOR_BLACK);
        init_pair(COLOR_GREEN, COLOR_GREEN, COLOR_BLACK);
        init_pair(COLOR_RED, COLOR_RED, COLOR_BLACK);
        init_pair(COLOR_CYAN, COLOR_CYAN, COLOR_BLACK);
        init_pair(COLOR_WHITE, COLOR_WHITE, COLOR_BLACK);
        init_pair(COLOR_MAGENTA, COLOR_MAGENTA, COLOR_BLACK);
        init_pair(COLOR_BLUE, COLOR_BLUE, COLOR_BLACK);
        init_pair(COLOR_YELLOW, COLOR_YELLOW, COLOR_BLACK);

        for (n = 1; n <= 8; n++) {
            attron(COLOR_PAIR(n));
            printw("color pair %d in NORMAL mode\n", n);
            attron(COLOR_PAIR(n) | A_STANDOUT);
```

```

        printw("color pair %d in STANDOUT mode\n", n);
        attroff(A_STANDOUT);
        refresh();
    }
    sleep(10);
} else {
    printw("Terminal does not support color\n");
    refresh();
    sleep(3);
}

app_exit();
exit(0);
}

```

外部条件（第 17 行以及第 41、44 行）保证程序在往下执行之前终端要支持彩色，而如果不支持彩色则能够正常退出。在初始化彩色系统之后（第 18 行），我们使用 `init_pair` 调用做了一些简单的色彩赋值（第 22~29 行），它将一对前景色和背景色与一个 `COLOR_PAIR` 类型的名字结合在一起：

```
int init_pair(short pair, short f, short b);
```

上面一行程序将前景色 `f` 和背景色 `b` 与 `pair` 结合起来。如果调用成功，则返回 `OK`；否则返回 `ERR`。

一旦完成了色彩的分配，我们就使用每一组颜色对（color pair）名以普通文本和标准输出模式显示文本（第 31~37 行）。这里我们使用 `attron` 调用来直接设置当前窗口（本例中为 `stdscr`）的显示属性（第 32 行和第 34 行），而不是为每个要显示的字符分别设置它的显示属性。在使用下一组颜色对之前，我们使用 `attroff` 调用关闭标准输出模式（第 36 行）。

```
int attron(int attrs);
int attroff(int attrs);
```

`attrs` 可以是一个或多个颜色和视属性经过逻辑“或”运算后的组合。

通常，`ncurses` 都带有一个用来控制窗口显示属性的扩展函数集合。它们在用户手册页 `curl_attr(3)` 中有详尽的记录。用户手册页 `curl_color(3)` 以详尽的篇幅讨论了 `ncurses` 的颜色处理接口。

23.8 窗口管理

前面我们已经涉及了一些 `ncurses` 的窗口管理函数。在这一节中，我们将对它们进行更详尽的讨论。所有的窗口管理例程都记录在用户手册页 `curl_window(3)` 中。

```
WINDOW *newwin(int nlines, int nclos, int begin_y, int begin_x);
int delwin(WINDOW *win);
WINDOW *subwin(WINDOW *orig, int nlines, int ncols, int begin_y,
```

```

        int begin_x);
WINDOW *derwin(WINDOW *orig, int nlines, int ncols, int begin_y,
               int begin_x);
WINDOW *dupwin(WINDOW *win);

```

`newwin` 创建并返回一个指向新建窗口的指针，该新建窗口具有 `ncols` 列和 `nlines` 行。其左上角位于坐标位置 `begin_y, begin_x`。`subwin` 和 `derwin` 创建并返回指向具有 `ncols` 列和 `nlines` 行的新建窗口的指针，该新建窗口位于父窗口 `orig` 的中心位置。`subwin` 创建的子窗口左上角位于相对于屏幕的坐标位置 `begin_y, begin_x`，而非相对于父窗口。`derwin` 的功能和 `subwin` 很类似，只是它所创建的子窗口的左上角将位于相对于父窗口 `orig` 的坐标位置 `begin_y, begin_x`，而非相对于屏幕而言。`dupwin` 将创建父窗口 `orig` 的一个完全副本。`delwin` 函数在前面已经介绍过了。对于返回整数值的函数，在成功调用时将返回 `OK`，而调用失败时返回 `ERR`。而对于返回指针的函数，在调用失败时将返回空指针 `NULL`。前面的程序已经展示了其中一些函数的使用方法，因此我们将避开对它们使用的进一步阐述。

23.9 其他各种工具函数

`ncurses` 的实用例程给你带来了 `ncurses` 提供的各种功能，这包括低层 `ncurses` 函数，获取环境信息以及创建屏幕转储（`dump`）。

屏幕转储非常有趣。函数 `putwin` 将所有与窗口相关的数据写入文件中，而 `getwin` 则是从一个文件中读取所有与窗口相关的数据。

```

int putwin(WINDOW *win, FILE *filep);
WINDOW *getwin(FILE *filep);

```

`putwin` 将窗口 `win` 的所有数据拷贝到由 `filep` 指定的已打开文件中，而 `getwin` 则返回一个由 `filep` 中的内容创建的 `WINDOW` 的指针。

函数 `scr_dump` 和 `scr_restore` 和上面的两个函数完成相似的功能，只不过它们的操作对象是 `ncurses` 屏幕，而不是窗口。然而，你不能混合使用这些调用；试图用 `scr_restore` 来读取由 `putwin` 写入的数据将导致错误。

```

int scr_dump(const char *filename);
int scr_restore(const char *filename);

```

`filename` 是用来完成写或读操作的文件名。

关于在读或写屏幕转储时所使用的其他例程的信息，请参阅用户手册页 `curs_scr_dump(3)` 和 `curs_scr_util(3)`。

记录在 `curs_termattrs(3)`、`curs_termcap(3)` 和 `curs_terminfo(3)` 中的 `ncurses` 的 `terminfo` 和 `termcap` 例程，用来报告环境信息并使你可以直接访问 `termcap` 和 `terminfo` 数据库。下面的函数用于从 `terminfo` 和 `termcap` 数据库获取信息。

```

int baudrate(void);           以每秒的比特数 (bps) 为单位返回终端的输出速率
char erasechar(void);        返回用户当前的取消符 (erase character)

```

<code>char killchar(void);</code>	返回用户当前的删除字符 (kill character)
<code>char *termname(void);</code>	返回环境变量 \$TERM 的值, 仅截取前 14 个字符
<code>char *longname(void);</code>	返回终端属性的一个长的描述, 可以长达 128 个字符

除非你要对功能键编程或有其他的原因要直接访问终端数据库, 一般不需要直接访问 `termcap` 和 `terminfo` 数据库, 因此这里我们不讨论这些函数。

除了你所看到的 `getmaxyx` 函数调用, 还有三个例程被用来获取光标和窗口坐标: `getyx`、`getparyx` 和 `getbegyx`。 `getyx` 返回光标的坐标; `getbegyx` 返回窗口的起始坐标 (左上角坐标); 最后是 `getparyx` 函数, 当一个子窗口调用 `getparyx` 时, 返回调用窗口相对于父窗口的起始坐标。

```
void getyx(WINDOW *win, int y, int x);
void getbegyx(WINDOW *win, int y, int x);
void getparyx(WINDOW *win, int y, int x);
```

和 `getmaxyx` 一样, 这三个函数调用都是宏定义, 因此 `y` 和 `x` 在传递时不需要带 “&” 符号。我们最后的实例程序清单 23.9 显示了如何使用这些实用函数中的一部分。

程序清单 23.9 cursutil.c

```
/*
 * Listing 23.9
 * cursutil.c - curses utility routines
 */
#include <stdlib.h>
#include <unistd.h>
#include <curses.h>
#include <unctrl.h>
#include <errno.h>
#include "utilfcns.h"

int main(void)
{
    WINDOW *win;
    FILE *fdump;
    int xmax, ymax, n = 0;
    app_init();

    if (!has_colors()) {
        printw("Terminal does not support color\n");
        refresh();
        sleep(3);
        app_exit();
    }
    if (start_color() == ERR)
        err_quit("start_color");

    init_pair(COLOR_RED, COLOR_RED, COLOR_BLACK);
```

```

init_pair(COLOR_YELLOW, COLOR_YELLOW, COLOR_BLACK);
init_pair(COLOR_WHITE, COLOR_WHITE, COLOR_BLACK);

bkgd('#' | COLOR_PAIR(COLOR_RED));
refresh();
sleep(3);

if ((win = subwin(stdscr, 10, 10, 0, 0)) == NULL)
    err_quit("subwin");
wbkgd(win, '@' | COLOR_PAIR(COLOR_YELLOW));
wrefresh(win);
sleep(1);

getmaxyx(stdscr, ymax, xmax);
while (n < xmax - 10) {
    mvwin(win, ((ymax - 10) / 2), n);
    refresh();
    sleep(1);
    if (n == 20) {
        /* Dump the subwindow to a file */
        fdump = fopen("dump.win", "w+");
        putwin(stdscr, fdump);
        fclose(fdump);
    }
    n += 10;
}

fdump = fopen("dump.win", "r");
win = getwin(fdump);
wrefresh(win);
sleep(3);

clear();
bkgd(' ' | COLOR_PAIR(COLOR_WHITE));
mvprintw(1, 1, "ERASE character: %s\n", unctrl(erasechar()));
mvprintw(2, 1, "KILL character : %s\n", unctrl(killchar()));
mvprintw(3, 1, "BAUDRATE (bps) : %d\n", baudrate());
mvprintw(4, 1, "TERMINAL type : %s\n", termname());
refresh();
sleep(5);

delwin(win);
app_exit();
exit(0);
}

```

我们在这个例子中加入了许多内容。第 20~27 行确保终端支持彩色，如果不支持程序就退出执行。接下来的三行代码（第 29~31 行）设置我们在程序后面将要用到的颜色值。在将背景色设置为黑色带有红色的“#”号之后（第 33~35 行），我们创建了一个 10×10

的子窗口，并且设置其背景色为黑底色带有黄色的“@”好。然后，我们以 10 个单位为增幅移动子窗口穿过其父窗口（第 43~55 行），并在 `n=20` 处创建一个窗口转储。为了说明重新显示一个经转储的窗口是很简单的，我们在第 57~60 行读取转储数据。最后，一旦我们清除了 `stdscr`，就将窗口设置为白色，背景为黑色，并使用终端信息例程来显示有关当前窗口显示环境的少量信息（第 62~69 行）。在删除子窗口后，程序结束。

23.10 小 结

如果顺利的话，本章通过演示了足够多的 `ncurses` 功能使读者清楚地认识到：`ncurses` 是一种比 `termios` 容易使用且更加强大的用于控制显示的接口。`ncurses` 是一个非常普遍的函数库，被许多流行的应用程序所使用，比如邮件客户程序 `mutt`、Midnight Commander 文件管理器、`lynx`（基于文本的浏览器）、`ncftp`（文件传输程序）和 `nvi` 等。

第 24 章 ncurses 高级编程

这一章专门介绍 ncurses 更高级的特性。前一章介绍的 ncurses 的功能也存在于其他 curses 库中，能够为大量类型的终端所支持。下面的例程能在一些而不是全部的其他 curses 库中使用。虽然这些例程能让基于字符的用户界面设计起来更容易，但是它们可能存在移植性问题。在使用这些例程之前，要确认你的所有目标平台和终端都能够支持它们的特性。这对于鼠标函数来说尤其如此。

虽然知道有上述警告存在，但这些例程对于在很短的时间内构造一个更加现代的界面确实有上佳的表现。

24.1 其他 ncurses 功能

基本的 curses 库提供的例程能够控制屏幕并且获得用户从键盘的输入。对于简单任务来说这就足够了，但是对于引入了图形化用户界面的情况来说，在各种应用之间有某种元素是相同的。ncurses 提供的函数中有许多支持鼠标、菜单和窗体的 API。

为了充分利用这些程序，你应该组织好它们以便使用一个事件循环。你的程序应该等待用户输入，然后根据报告的动作执行一项功能。对于没有用过 curses、基于文本编程的程序员来说这是一种不熟悉的方法，而对于那些开发过基于图形用户界面的应用的程序员来说就非常熟悉了。

24.1.1 鼠标支持

对于支持鼠标的终端来说，鼠标是一种用于获得某种类型的用户输入的出色工具。超链接可能是当前使用鼠标的最显著的例子，但是还有其他用户界面元素能够得益于鼠标的使用。你还可以通过鼠标让单选按钮（radio button）、复选框（check box）和其他标准的用户界面元素比在没有鼠标的、基于字符的用户界面上更容易使用。但要记住，只有一定数量的终端能够支持鼠标，所以总要有使用键盘的替代方案。

24.1.2 菜单支持

菜单支持能够让你很容易地创建一个可供选择的列表。根据菜单驱动程序处理布局任务和获得用户输入的方式，可以有多种风格的菜单可供选择。

24.1.3 窗体支持

窗体库能够让你创建复杂的窗体而不必自己完成所有的屏幕更新工作。通过一点编程工作，它还能对数据做有效性检查。

24.2 和鼠标交互

现代的图形用户界面都使用鼠标。因为大多数终端实际上都是某种形式的 `xterm`，所以基于字符的用户界面也支持鼠标。`ncurses` 通过一个非常简单而又功能强大的 API 提供在 `xterm` 上的鼠标支持。`ncurses` 随带的文档警告说这个 API 可能会有变化，但似乎它涵盖了指示设备能有的大多数用法。

24.2.1 鼠标 API 概述

鼠标 API 很简单，而且如果程序使用了一个事件循环，并由 `getch` 来驱动程序的执行，那么就能很容易地把鼠标 API 集成到这个已有的程序之中。终端必须在非规范模式下接受鼠标事件。`ncurses` 例程模拟了规范模式，比如 `getstr`（这个函数在结束之前期望得到一行输入），它会阻塞鼠标事件。在那样的情况下会产生一声蜂鸣来报错。另外，你应该有功能键，因为没有它 `xterm` 不会正确地报告鼠标事件。这些任务可以用下面的代码来完成：

```
raw();
keypad(stdscr, TRUE);
```

如果你要把 `getch` 作为对输入队列的一种操作，那么鼠标 API 则引入了另一种队列——鼠标队列。鼠标队列能让你知道在输入队列中有鼠标事件等待处理，然后你就能用鼠标 API 例程访问鼠标队列。

鼠标 API 引入了一种新的数据类型和一些新的常量。事先知道它们是会有帮助的。第一个是 `KEY_MOUSE`。鼠标事件本身通过 `MEVENT` 类型捕获，这个类型的定义如下：

```
typedef struct {
    short id;           /* ID to distinguish multiple devices */
    int x, y, z;        /* event coordinates */
    mmask_t bstate;     /* button state bits */
} MEVENT;
```

坐标 `z` 目前尚未用到，而是供将来使用。`id` 域用于区分多个设备，比如一个图形板和一个鼠标。`x` 和 `y` 给出了事件发生时鼠标的位置。`bstate` 域则显示了事件发生时鼠标按键的状态。表 24.1 列出了用来决定这一信息的常量。它们还用来告诉 `ncurses` 你对什么事件感兴趣。

表 24.1 事件常量

事件名称	描述
<code>BUTTON1_PRESSED</code>	按下鼠标按键 1
<code>BUTTON1_RELEASED</code>	松开鼠标按键 1
<code>BUTTON1_CLICKED</code>	单击鼠标按键 1
<code>BUTTON1_DOUBLE_CLICKED</code>	双击鼠标按键 1
<code>BUTTON1_TRIPLE_CLICKED</code>	连续点击 3 次鼠标按键 1
<code>BUTTON2_PRESSED</code>	按下鼠标按键 2

(续表)

事件名称	描述
BUTTON2_RELEASED	松开鼠标按钮 2
BUTTON2_CLICKED	单击鼠标按钮 2
BUTTON2_DOUBLE_CLICKED	双击鼠标按钮 2
BUTTON2_TRIPLE_CLICKED	连续点击 3 次鼠标按钮 2
BUTTON3_PRESSED	按下鼠标按钮 3
BUTTON3_RELEASED	松开鼠标按钮 3
BUTTON3_CLICKED	单击鼠标按钮 3
BUTTON3_DOUBLE_CLICKED	双击鼠标按钮 3
BUTTON3_TRIPLE_CLICKED	连续点击 3 次鼠标按钮 3
BUTTON4_PRESSED	按下鼠标按钮 4
BUTTON4_RELEASED	松开鼠标按钮 4
BUTTON4_CLICKED	单击鼠标按钮 4
BUTTON4_DOUBLE_CLICKED	双击鼠标按钮 4
BUTTON4_TRIPLE_CLICKED	连续点击 3 次鼠标按钮 4
BUTTON_SHIFT	Shift 键+鼠标按钮的状态改变
BUTTON_CTRL	Control 键+鼠标按钮的状态改变
BUTTON_ALT	Alt 键+鼠标按钮的状态改变
ALL_MOUSE_EVENTS	报告所有鼠标按钮的状态改变
REPORT_MOUSE_POSITION	报告鼠标移动

最后两个事件只用来告诉 ncurses 你所感兴趣的事件是什么。

24.2.2 鼠标控制例程

在从鼠标获得信息之前，你需要告诉 ncurses 你对什么样的鼠标事件感兴趣。完成这一功能的函数叫作 `mousemask`，它的原型如下：

```
mmask_t mousemask(mmask_t newmask,mmask_t *oldmask);
```

你可以使用表 24.1 列出的事件填充 `newmask` 和 `oldmask` 两个参数。函数执行出错则返回 0，执行成功就返回掩码的当前设置。这个函数的起始调用可能与下面类似：

```
if (!mousemask(BUTTON1_CLICKED | BUTTON2_CLICKED |
    BUTTON3_CLICKED )) {
   printw("Failed to initialise mouse!");
}
```

在这个例子里，ncurses 被告知当单击鼠标按钮 1~3 时产生鼠标事件。一旦 `mousemask` 成功返回，当有鼠标事件正在等待处理时，ncurses 的 `getch` 系列函数调用就返回 `KEY_MOUSE`。再次调用 `getch` 之前，你必须获得鼠标事件，否则它就会丢失。有两个例程用来操作鼠标事件队列。它们的原型如下：

```
int getmouse(MEVENT *event);
int ungetmouse(MEVENT *event);
```

`getmouse` 用来获得下一个鼠标事件。如果出错它将返回 `ERR`，如果执行成功则返回 `OK`。另外，如果执行成功它将填充传递给它的 `MEVENT` 结构。`ungetmouse` 函数会把事件返回给鼠标事件队列，也会把 `KEY_MOUSE` 事件返回给 `getch` 函数正在工作的输入队列。

一次鼠标点击定义为在一定时间内一个按键被按下又被释放。默认的时间是 1/5 秒，但可以用 `mouseinterval` 函数调整这个时间。这个函数有一个参数，是检测鼠标点击所允许的时间，单位是千分之一秒。同样，这个函数在执行成功时返回 `OK`，失败时返回 `ERR`。

```
int mouseinterval(int erval);
```

最后两个工具函数处理 `MEVENT` 结构返回的 `x` 和 `y` 坐标。`wenclose` 用来判断传递给它的 `y` 和 `x` 值是否落在给定的窗口中。`getmouse` 返回的坐标是相对于屏幕的（可以和 `stdscr` 相同，但不必一定相同）值，所以要使用 `wenclose` 来检测在你创建的子窗口中是否有鼠标事件发生。如果在子窗口中发生了鼠标点击事件，你就可以使用 `wmouse_trafo` 函数产生相对于子窗口的屏幕坐标。把你要用的窗口以及要转换的坐标作为参数传递给这个函数。最后一个参数应该为 `FALSE`。为了把相对于子窗口的坐标转换为相对于屏幕的坐标，则要把最后一个参数设置为 `TRUE`。为了方便起见，宏 `mouse_trafo` 调用了第一个参数设为 `stdscr` 的 `wmouse_trafo` 函数，这个宏遵循了正常的 `curses` 命名规范。函数的原型如下：

```
bool wenclose(WINDOW *win, int y, int x);
bool wmouse_trafo(const WINDOW *win, int* pY, int* pX, bool
to_screen);
```

24.2.3 示例程序

程序清单 24.1 是一个简单的程序，它能够检测是否使用了鼠标，并报告鼠标事件。它还演示了那些事件在子窗口的某些位置发生时程序不同的执行行为。

程序清单 24.1 chap24.1.c

```
/*
 * Listing 24.1
 */
#include <stdlib.h>
#include <unistd.h>
#include <curses.h>
#include <errno.h>
#include "utilfcns.h"

int
main(int argc, char *argv[])
{
    int quit = 0, key;
    WINDOW *list;
#ifdef NCURSES_MOUSE_VERSION
    MEVENT mevent;
```

```

#endif

    app_init();

#ifdef NCURSES_MOUSE_VERSION
    mvaddstr(0, 0, "Mouse demo can't run - NCURSES_MOUSE_VERSION"
              " is not defined.");
    refresh();
    sleep(3);
#else
# if NCURSES_MOUSE_VERSION != 1
    mvaddstr(0, 0, "Mouse demo can't run - wrong version of"
              " NCURSES_MOUSE_VERSION.");
    refresh();
    sleep(3);
# else
    raw();
    keypad(stdscr, TRUE);
    if (mousemask(BUTTON1_CLICKED|BUTTON2_PRESSED|
                 BUTTON2_RELEASED, 0) == 0) {
        mvaddstr(0, 0, "Mouse demo can't run - terminal doesn't"
                    " support mouse");
        refresh();
        sleep(3);
    } else {
        list = subwin(stdscr, 5, 16, 9, 31);
        wborder(list, '|', '|', '-', '-', '+', '+', '+', '+');
        mvwaddstr(list, 1, 1, "Cheddar");
        mvwaddstr(list, 2, 1, "swiss");
        mvwaddstr(list, 3, 1, "guda");
        mvaddstr(0, 0, "Mouse demo, press q to quit.");
        refresh();

        while (!quit) {
            key = getch();
            switch (key) {
                case KEY_MOUSE:
                    getmouse(&mevent);
                    if (wenclose(list, mevent.y, mevent.x)) {
                        wmouse_trafo(list, &mevent.y, &mevent.x,
                                    FALSE);
                        if (mevent.x > 0 && mevent.x < 15) {
                            switch (mevent.y) {
                                case 0:
                                case 4:
                                    mvprintw(1, 0, "Missed! ");
                                    break;
                                case 1:

```

```

        mvprintw(1, 0, "Cheddar! ");
        break;
    case 2:
        mvprintw(1, 0, "Swiss! ");
        break;
    case 3:
        mvprintw(1, 0, "Guda! ");
        break;
    default:
        mvprintw(1, 0, "Huh? ");
        break;
    }
} else {
    mvprintw(1, 0, "Missed! ");
}
} else {
    mvprintw(1, 0, "Squeek! ");
}
printw("(z:%d, y:%d, x:%d", mevent.z,
        mevent.y, mevent.x);
if (mevent.bstate & BUTTON1_CLICKED) {
    printw(", BUTTON1_CLICKED");
} else if (mevent.bstate & BUTTON2_PRESSED) {
    printw(", BUTTON2_PRESSED");
} else if (mevent.bstate & BUTTON2_RELEASED) {
    printw(", BUTTON2_RELEASED");
} else {
    printw(", huh?");
}
clrtoeol();
refresh();
break;
case 'q':
    quit = 1;
    break;
default:
    mvaddstr(1, 0, "No, press use the mouse
                or press q.");
    clrtoeol();
    refresh();
    break;
}
}
}
# endif
#endif
/* NCURSES_MOUSE_VERSION - wrong version */
/* NCURSES_MOUSE_VERSION - doesn't exist */

```

```
    app_exit();  
    exit(0);  
}
```

这个程序演练了大多数鼠标例程。第 15 行和第 21 行检查以确保我们正在使用的 `ncurses` 库支持鼠标。第 27 行确认它的版本为 1；文档警告说鼠标 API 可能在今后有所改变。对于这个例子来说，如果程序发现没有鼠标存在或者鼠标不起作用（第 35 行），程序就中止执行，但是在一个实际应用中即使没有鼠标支持也应该继续执行。第 33~35 行设置鼠标。如果第 35 行执行成功，我们就进入了事件循环（第 49 行）并等待一个事件发生（第 50 行）。通过 `switch` 语句，我们检测到一个 `KEY_MOUSE` 事件并得到了这个鼠标事件（第 52~53 行）。在第 53~54 行中，我们判断鼠标事件发生的位置，而在第 82~90 行判断发生了什么样的鼠标事件。

24.3 使用菜单

`ncurses` 菜单库提供了一种简单的可配置的系统，用来提供菜单项列表供用户从中进行选择。编译和链接支持菜单功能的 `ncurses` 程序和正规的 `ncurses` 程序类似，但是需要在菜单例程或常量的任何文件中包含头文件 `menu.h`。在链接过程中，需要在链接命令行加入 `-lmenu`。虽然按照这样的顺序：`-lmenu -lncurses` 把 `-l` 选项传递给链接器对于 Linux 来说并不是必须的，但这样做可以增强对其他操作系统的可移植性。在某些系统上，链接器只对库做一遍扫描，从库中去除它还没有看到在程序使用的例程。因为菜单库使用了大量 `ncurses` 库中的函数，偶尔的情况下，它也会用到一个你的程序中没有用到的 `ncurses` 例程，因此在那个系统上就会导致链接失败。

和鼠标例程类似，菜单也跟随稍有些复杂的输入循环。程序首先创建菜单项的列表，然后再创建整个菜单。和鼠标例程一样，程序使用 `getch` 检索输入，但是在不可打印的字符被转换为菜单移动命令之后，是由一个叫作 `menu_driver` 的函数来处理输入的。最后，当不再需要菜单时，就释放菜单项和菜单。

24.3.1 菜单 API 概述

Menu API（菜单 API）可以分成 3 个部分：管理菜单项的例程、管理整个菜单的例程以及处理输入的例程。另外，还有一些例程负责处理某些额外的选项和钩子。

菜单库能够把你从在屏幕上构造菜单的工作中解放出来，但是仍需要你告诉库为你构造什么样的菜单。首先，你要在一个数组中定义菜单项的集合。对于每个所需的菜单，你都需要一个菜单项数组。一旦创建好了数组，就有例程能够从菜单项中取得信息，比如名称和描述，也有例程能设置选项，还有例程能取得或设置菜单项的属性。

还有多种例程能操作整个菜单。其中用得最多的例程是创建、撤销菜单的例程以及显示、隐藏菜单的例程。需要指出重要的一点，和基本的 `ncurses` API 一样，菜单显示例程也不能立即在屏幕上显示菜单；应调用 `wrefresh` 或 `refresh` 来做到这一点。

一个菜单实际上由两个窗口组成（默认情况下都是 `stdscr`）。菜单使用外面的窗口构成菜单的边界和标题以及其他需要的部分；菜单例程实际上并不触及这个窗口。内部的窗口是绘制菜单的地方。有几个例程能够控制菜单的大小以及它们本身的屏幕。

通常使用 `getch` 检索输入，然后交由 `menu_driver` 函数处理。通过移动、切换或者采用模式缓冲操作的菜单项会被加亮显示。后者能够让用户键入一个菜单项的开始部分，然后加亮与之匹配的菜单项。有时，可能需要确定当前指向的是哪个特定菜单项；另一个例程能够处理这种需求。而且你还可以在任何地方检索当前选中的菜单项。

最后，有一组例程能够控制菜单的外观和功能。这里有 3 个函数的手册页面较重要。`menu_opts` 页面介绍了用于控制菜单外观和功能的例程和常量。在程序中可以加入钩子函数，以便在初始化菜单、菜单项选择变化或者删除菜单时可以执行某些动作；这些都在 `menu_hooks` 中进行介绍。最后，菜单项保存的内容可以不只是名称和描述；它们还能够保存指针，指向针对应用的特定数据。这会在 `item_userptr` 中介绍。这些例程都不是严格需要的，但是使用它们可以简化应用程序的操作，或者让应用程序对用户更友好。

24.3.2 菜单控制例程

设立菜单的第一步是调用 `new_item` 创建菜单项。创建菜单的例程需要一个菜单项的数组，所以如果菜单项创建在 `ITEM *` 类型的数组中，最好给数组的最后一项赋 `NULL` 值。使用 `free_item` 函数能够释放菜单项，这个函数接收一个 `ITEM *` 变量作为释放的菜单项。

使用下面的例程创建和操作菜单项：

```
int free_item(ITEM *item)
```

释放分配给一个 `ITEM *` 变量的内存。

```
ITEM *new_item(char *name, char *desc)
```

给一个 `ITEM *` 结构分配空间并且进行初始化。

```
OPTIONS item_opts(const ITEM *item)
```

返回当前设置的选项位。目前只有一个选项：`O_SELECTABLE`，它控制一个菜单项是否被选中。默认设置这个选项。

```
int item_opts_off(ITEM *item, OPTIONS opts)
```

这个函数关闭传递给它的选项，保持其他选项不动。

```
int item_opts_on(ITEM *item, OPTIONS opts)
```

这个函数打开传递给它的选项，保持其他选项不动。

```
int set_item_opts(ITEM *item, OPTIONS opts)
```

把菜单项的选项设置为传递给它的选项。

```
void *item_userptr(ITEM *item)
```

返回和 `userptr` 关联的菜单项。

```
int set_item_userptr(ITEM *item)
```

为菜单项设置 userptr。

```
bool item_value(ITEM *item)
```

对于可以选中多项的菜单（那些关闭 O_ONEVALUE 设置的菜单），如果菜单项被选中则返回 TRUE 或 FALSE。

```
int set_item_value(ITEM *item, bool value)
```

用于选中或不选中可多选菜单中的一个菜单项。

```
bool item_visible(ITEM *item)
```

用于判断菜单项当前是否可见——用于菜单太长超出它的窗口而需要滚动显示的情况。

一旦创建并配置好了菜单项，就需要创建菜单。一个菜单包含了菜单项的一个集合，并且把菜单项显示给用户。正像菜单项一样，菜单也有许多能够配置它们的选项。

下面的函数创建并控制整个菜单：

```
MENU *new_menu(ITEM **items)
```

创建一个新菜单，items 必须是以 NULL 结尾的数组。

```
int free_menu(MENU *menu)
```

释放分配给一个菜单的内存。

```
int post_menu(MENU *menu)
```

绘制一个菜单，但在屏幕上不显示它。使用 wrefresh 显示菜单。

```
int unpost_menu(MENU *menu)
```

从子窗口中擦除菜单；所做的改动不在屏幕上显示。使用 wrefresh 来显示改动。

```
int item_count(const MENU *menu)
```

返回菜单中菜单项的数目。

```
ITEM **menu_items(const MENU *menu)
```

返回菜单中的菜单项列表。

```
int set_menu_items(const MENU *menu, ITEM **items)
```

用传递给菜单的菜单项改变菜单。只能用于已改变但还未显示的菜单（unpost）。

```
OPTIONS menu_opts(MENU *menu, OPTIONS opts)
```

把传递给菜单的选项 opts 都设置为关闭，保持其他选项不动。所有的选项默认都设置为打开；可能的选项有：

- O_ONEVALUE——菜单中只能选中一项

- `O_SHOWDESC`——显示菜单项的描述
- `O_ROWMAJOR`——以行的顺序显示菜单
- `O_IGNORECASE`——当匹配模式时忽略大小写
- `O_SHOWMATCH`——移动光标指示模式匹配的菜单名
- `O_NONCYCLIC`——在菜单的末尾不折叠显示

```
int menu_opts_off(MENU *menu, OPTIONS opts)
```

把菜单上的选项设置为 `opts` 传递的值。

```
int menu_opts_on(MENU *menu, OPTIONS opts)
```

把传递给菜单的选项 `opts` 都设置为打开，并且保持其他选项不动。

```
int set_menu_opts(MENU *menu)
```

返回当前选项设置。

```
int scale_menu(MENU *menu, int *row, int *cols)
```

返回用于菜单的最小子窗口的大小。

```
int set_menu_sub(MENU *menu, WINDOW *sub)
```

设置和菜单相关联的子窗口。如果 `sub` 为 `NULL`，则菜单相关联的子窗口就设置为 `stdscr`。

```
int set_menu_win(MENU *menu, WINDOW *win)
```

设置和菜单相关联的窗口。如果 `win` 为 `NULL`，则该窗口就是 `stdsrc`。

```
WINDOW *menu_sub(const MENU *menu)
```

返回菜单使用的子窗口。

```
WINDOW *menu_win(const MENU *menu)
```

返回和菜单相关联的窗口。

```
chtype menu_back(const MENU *menu)
```

返回可选但未选中的菜单项的属性，默认为 `A_NORMAL`。

```
chtype menu_fore(const MENU *menu)
```

返回选中的菜单项的属性，默认为 `A_STANDOUT`。

```
chtype menu_grey(const MENU *menu)
```

返回未选中的菜单项的属性，默认为 `A_UNDERLINE`。

```
int menu_pad(const MENU *menu)
```

返回用来把菜单名和菜单描述分隔开的字符，默认为空白。

```
int set_menu_back(const MENU *menu, chtype attr)
```


设置可选但未选中菜单项的属性，默认为 A_NORMAL。

```
int set_menu_fore(const MENU *menu, chtype attr)
```

设置选中菜单项的属性，默认为 A_STANDOUT。

```
int set_menu_grey(const MENU *menu, chtype attr)
```

设置未选中菜单项的属性，默认为 A_UNDERLINE。

```
int set_menu_pad(const MENU *menu, int pad)
```

设置用来把菜单名和菜单描述分隔开的字符，默认为空白。

```
int set_menu_format(const MENU *menu, int rows, int cols)
```

设置当前菜单的显示大小。如果菜单包含的菜单项比给出的空间大，则滚动显示。

```
int menu_format(const MENU *menu, int rows, int cols)
```

返回当前菜单的显示大小。

```
int set_menu_mark(const MENU *menu, const char *mark)
```

某些终端不能加亮显示菜单项选择情况。这个函数设置一个前缀字符串以指出当前菜单项的选择。默认的标记为-。

```
const char *menu_mark(const MENU *menu)
```

返回当前的前缀字符串。

```
int menu_request_by_name(const char *name)
```

返回和 **name** 相关联的请求代码，如果没有找到则返回 E_NO_MATCH。

```
const char *menu_request_name(int request)
```

返回 **request** 可打印的名字。

```
int set_menu_spacing(const MENU *menu, int spc_desc, int spc_rows,
int spc_cols)
```

选择菜单使用的间隔空间。在菜单名和菜单描述之间的间隔由 **spc_desc** 控制；**spc_rows** 和 **spc_cols** 控制菜单项单行和单列的间隔空间。**spc_rows** 不能超过 3；其他值必须小于 TABSIZE。

```
int menu_spacing(const MENU *menu, int spc_desc, int spc_rows,
int spc_cols)
```

返回菜单的间隔空间设置。

```
void *menu_userptr(const MENU *menu)
```

返回和菜单相关的针对应用的特定数据。菜单代码不会影响这些数据。

```
int set_menu_userptr(const MENU *menu, void *userptr)
```

设置针对应用的特定数据。

最后，还有处理用户输入的例程。处理输入的主要函数是下面介绍的 `menu_driver`，它采用表 24.2 中的常量来描述不同的用户动作。

表 24.2 `menu_driver` 使用的常量

事件名称	描述
<code>REQ_LEFT_ITEM</code>	向左移动到一个菜单项
<code>REQ_RIGHT_ITEM</code>	向右移动到一个菜单项
<code>REQ_UP_ITEM</code>	向上移动到一个菜单项
<code>REQ_DOWN_ITEM</code>	向下移动到一个菜单项
<code>REQ_SCR_ULINE</code>	向上滚动一行
<code>REQ_SCR_DLINE</code>	向下滚动一行
<code>REQ_SCR_DPAGE</code>	向下滚动一页
<code>REQ_SCR_UPAGE</code>	向上滚动一页
<code>REQ_FIRST_ITEM</code>	移动到第一个菜单项
<code>REQ_LAST_ITEM</code>	移动到最后一个菜单项
<code>REQ_NEXT_ITEM</code>	移动到下一个菜单项
<code>REQ_PREV_ITEM</code>	移动到上一个菜单项
<code>REQ_TOGGLE_ITEM</code>	选中/不选中一个菜单项
<code>REQ_CLEAR_PATTERN</code>	清除菜单模式缓冲
<code>REQ_BACK_PATTERN</code>	从模式缓冲里删除前一个字符
<code>REQ_NEXT_MATCH</code>	移动到匹配模式的下一个菜单项

菜单函数并不直接读入用户的输入。这是由 `getch` 处理的，上一章已经介绍过这个函数。当击键行为被转换为一个动作或者被判断出是一个可打印的字符之后，需要使用下面的例程：

```
int menu_driver(MENU *menu, int c)
```

参数 `c` 的传递值不是一个可打印的字符就是表 24.2 中列出的函数的常量。根据需要重画菜单，以便反馈给用户，告诉他们正在执行什么操作。可打印的字符被加入到模式缓冲当中，而与之最接近的菜单项则被选中。

```
const char *item_description(const ITEM *item)
```

返回和参数指定的菜单项相关联的菜单项描述。

```
const char *item_name(const ITEM *item)
```

返回和参数指定的菜单项相关联的菜单项名称字符串。

```
ITEM *current_item(const MENU *menu)
```

返回当前选中的菜单项。

```
int item_index(const ITEM *item)
```

返回菜单项在菜单项列表中的位置。

```
int pos_menu_cursor(const MENU *menu)
```

把光标定位在当前选中的菜单项上。正常情况下不需要这样做，但是如果调用了其他 `curses` 函数可能会需要。

```
inet set_current_item(MENU *menu, const ITEM *item)
```

把 `menu` 菜单里当前选中的菜单项设置为 `item`。

```
int set_top_row(MENU *menu, int row)
```

尝试把 `row` 设置为显示的最顶端一行。

```
int top_row(MENU *menu)
```

返回显示的最顶端菜单项的行数。

```
char *menu_pattern(const MENU *menu)
```

返回模式缓冲的当前内容。

```
int set_menu_pattern(MENU *menu, char *pattern)
```

把模式缓冲的当前内容设置为 `pattern`。

24.3.3 示例程序

程序清单 24.2 显示了一个用于向屏幕打印输出的菜单。通过按 `Enter` 键来选择它们，然后用方向键在它们中间移动。这是一个简单的例子，但是它用到了主要的菜单例程，给你提供了一个试验上述函数的界面。

程序清单 24.2 chap24.2.c

```
/*
 * Listing 24.2
 */
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <ctype.h>
#include <curses.h>
#include <menu.h>
#include <errno.h>
#include "utilfcns.h"

char *say_menu[] = {"Hi", "Say Hi",
                    "World", "Hello World",
                    "Quit", "Exit the Application",
                    NULL};

int
```

```
main(int argc, char *argv[])
{
    int quit = 0, key, i;
    MENU *menu;
    ITEM *items[4];

    app_init();

    for (i = 0; say_menu[i*2]; i++) {
        items[i] = new_item(say_menu[i*2], say_menu[(i*2)+1]);
    }
    items[i] = NULL;
    menu = new_menu(items);
    post_menu(menu);
    raw();
    noecho();
    keypad(stdscr, TRUE);
    while (!quit) {
        int code;
        ITEM *selected;

        key = getch();
        switch (key) {
            case KEY_DOWN:
            case KEY_RIGHT:
                code = menu_driver(menu, REQ_NEXT_ITEM);
                break;
            case KEY_UP:
            case KEY_LEFT:
                code = menu_driver(menu, REQ_PREV_ITEM);
                break;
            case '\n':
                if ((selected = current_item(menu)) != NULL) {
                    unpost_menu(menu);
                    mvaddstr(10, 10, "You have selected:");
                    mvaddstr(11, 15, item_description(selected));
                    refresh();
                    sleep(5);
                    erase();
                    if (!strcmp("Quit", item_name(selected))) {
                        quit=1;
                    } else {
                        post_menu(menu);
                    }
                }
                break;
            default:
                if (isprint(key)) {
```

```
        if (menu_driver(menu, key) != E_OK) {
            menu_driver(menu, REQ_CLEAR_PATTERN);
        }
    } /* else ignore it */
    break;
}
}

app_exit();
exit(0);
}
```

这个简单的例程勾勒出了菜单应用程序的一般框架，包括首先创建菜单项，然后把它们和一个菜单关联起来（在第 27~31 行），并且在 switch 语句中响应输入（第 41~72 行）。注意开始的 `curses` 调用让终端处于正确的状态：

```
raw();
noecho();
keypad(stdscr, TRUE);
```

这段代码让终端处于这样的状态，它能够响应每一次击键动作，但并不回显它。`keypad` 调用让 `getch` 例程能够对功能键（包括方向键）进行解码并把它们作为 `KEY_` 式的常量返回。

24.4 ncurses 窗体

一个窗体是用来接收并验证用户输入的域组成的一个列表。域的类型有几种——标签、数字字母域、IP 地址域以及用户自定义的域。和菜单类似，一个窗体也在两个窗口中。窗体库不影响主窗口；窗体主要在子窗口中工作。处理窗体的方式和处理菜单方式类似，区别在于，由于域的复杂性使得窗体库提供了更多的选项。

编译一个窗体应用程序需要链接 `curses` 库和窗体库。和菜单库类似，如果链接了窗体库那么最好把它放在 `ncurses` 库之前：

```
cc -o formapp formapp.c -lform -lncurses
```

而 `formapp.c` 应该包含下面两行：

```
#include <ncurses.h>
#include <form.h>
```

24.4.1 窗体 API 概述

Form API（窗体 API）一般和 Menu API 一样。但是，窗体 API 要大得多。一个窗体由两个窗口组成。外面的一个窗口是一个框架，窗体例程并不触及它。窗体例程使用子窗口。正如窗体可以和菜单类似一样，域也和菜单项类似，但是域的功能要大得多；这也是窗体库要大得多的地方。

域能够组成页。form_driver 例程扮演了和 menu_driver 相同的角色，不但能让按键在域之间移动也能在页之间移动。这可以让你创建由多个页构成的窗体，而且使用同样的事件循环处理和用户的交互工作。

但是，处理域的第一步工作是创建它。另外可以改变一个域的初始设置，而且可以设置域的多种属性。ncurses 有能够设置域的外观的例程，也有能够控制和一个域相关联的主缓冲和其他缓冲的例程，还有能够控制菜单库中菜单项所支持的其他用户数据的例程。域负责接受数据，从而就有设置数据验证方法的例程。

24.4.2 窗体管理例程

使用窗体库的第一步是创建域，然后再创建窗体。但是，因为控制 FORM 结构的 API 和控制 MENU 结构的 API 是如此相似，以至于这里可以接着上一节的内容继续介绍例程。Form API 的其余部分数量相当大，令人感到畏惧，所以最好能熟悉背景。

```
FORM *new_form(FIELD **fields)
```

用以 NULL 结尾的域列表中的域创建一个新窗体。

```
int free_form(FORM *form)
```

释放域数组并释放窗体占有的内存。

```
int post_form(FORM *form)
```

显示窗体。这个函数不会把窗体显示到屏幕上；使用 wrefresh 做到这一点。

```
int unpost_form(FORM *form)
```

从窗体所在的子窗口中擦除窗体。同样需要使用 wrefresh。

还有许多用于窗体选项的函数、和窗体相关联的用户数据以及和窗体相关的窗口的 ncurses 例程。要记住，窗体库不会触及窗口，它只是更新子窗口。默认情况下，窗体库使用 stdscr。它也不会触及用户数据；完全由你的程序来保存和释放 userptr 中的数据。

```
int set_form_opts(FORM *form, OPTIONS opts)
```

用参数 opts 传递的选项设置窗体选项。该选项为 O_NL_OVERLOAD 时，如果用户在域的末尾按了回车键，则强迫窗体驱动程序移动到下一个域；该选项为 O_BS_OVERLOAD 时，如果在域的开头按了退格键，则强迫窗体驱动程序移动到上一个域。这些选项默认情况下都是打开的。

```
int form_opts_on(FORM *form, OPTIONS opts)
```

把 opts 传递的选项在窗体上都设置为打开，保持其他选项不动。

```
int form_opts_off(FORM *form, OPTIONS opts)
```

把 opts 传递的选项在窗体上都设置为关闭，保持其他选项不动。

```
OPTIONS form_opts(const FORM *form)
```

返回当前设置的选项。

```
int set_form_sub(FORM *form, WINDOW *win)
```

设置窗体的子窗口。

```
WINDOW *form_sub(const FORM *form)
```

返回窗体的子窗口。

```
int set_form_win(FORM *form, WINDOW *win)
```

设置窗体的窗口。

```
WINDOW form_win(const FORM *form)
```

返回窗体的窗口。

```
int scale_form(const FORM *form, int *rows, int *columns)
```

返回要求一个窗口的最小尺寸。

```
int set_form_userptr(FORM *form, void *userptr)
```

为窗体设置针对应用的特定数据的指针

```
void *form_userptr(const FORM *form)
```

返回给定窗体的针对应用的特定数据。

除了创建窗体之外，你还要获得它们的输入。使用 `form_driver` 例程做到这一点。`form_driver` 例程比 `menu_driver` 例程有更多的移动功能，它们将在下面的表格中进行描述。`form_driver` 例程的原型如下：

```
int form_driver(FORM *form, int c)
```

另外，下面这两个函数既能返回一个请求的文字名称，也能把名称转换为一个请求代码。第二个函数如果执行失败返回 `E_NO_MATCH`。

```
const char *form_request_name(int request)
```

```
int form_request_by_name(const char *name)
```

表 24.3 列出了大量的移动路线。之所以有那么多移动路线主要和这样的事实有关：存在用来编辑文本的域。还有一些例程是用来移动到一个窗体的其他页或其他域上。

表 24.3 移动路线

事件名称	描述
REQ_NEXT_PAGE	移到下一页
REQ_PREV_PAGE	移到上一页
REQ_FIRST_PAGE	移到第一页
REQ_LAST_PAGE	移到最后一页
REQ_NEXT_FIELD	移到下一个域
REQ_PREV_FIELD	移到上一个域
REQ_FIRST_FIELD	移到第一个域

(续表)

事件名称	描述
REQ_LAST_FIELD	移到最后一个域
REQ_SNEXT_FIELD	移到排序的下一个域
REQ_SPREV_FIELD	移到排序的上一个域
REQ_SFIRST_FIELD	移到排序的第一个域
REQ_SLAST_FIELD	移到排序的最后一个域
REQ_LEFT_FIELD	移到左边一个域
REQ_RIGHT_FIELD	移到右边一个域
REQ_UP_FIELD	移到上一个域
REQ_DOWN_FIELD	移到下一个域
REQ_NEXT_CHAR	移到下一个字符
REQ_PREV_CHAR	移到上一个字符
REQ_NEXT_LINE	移到下一行
REQ_PREV_LINE	移到上一行
REQ_NEXT_WORD	移到下一个单词
REQ_PREV_WORD	移到上一个单词
REQ_BEG_FIELD	移到域开头
REQ_END_FIELD	移到域末尾
REQ_BEG_LINE	移到行开头
REQ_END_LINE	移到行末尾
REQ_LEFT_CHAR	在域内向左移
REQ_RIGHT_CHAR	在域内向右移
REQ_UP_CHAR	在域内向上移
REQ_DOWN_CHAR	在域内向下移
REQ_NEW_LINE	插入或覆盖一个新行
REQ_INS_CHAR	在光标处插入一个空白
REQ_INS_LINE	在光标处插入一个空白行
REQ_DEL_CHAR	删除光标处的字符
REQ_DEL_PREV	删除光标之前的字符
REQ_DEL_LINE	删除光标所在行
REQ_DEL_WORD	删除光标当前位置以空白结尾的单词
REQ_CLR_EOL	清除从光标当前位置到行尾的内容
REQ_CLR_EOF	清除从光标当前位置到域结尾的内容
REQ_CLR_FIELD	清除整个域
REQ_OVL_MODE	进入覆盖模式
REQ_INS_MODE	进入插入模式
REQ_SCR_FLINE	在域内向前滚动一行

(续表)

事件名称	描述
REQ_SCR_BLINE	在域内向后滚动一行
REQ_SCR_FPAGE	在域内向前滚动一页
REQ_SCR_BPAGE	在域内向后滚动一页
REQ_SCR_FHPAGE	在域内向前滚动半页
REQ_SCR_BHPAGE	在域内向后滚动半页
REQ_SCR_FCHAR	在域内向前滚动一个字符
REQ_SCR_BCHAR	在域内向后滚动一个字符
REQ_SCR_HFLINE	在域内水平向前滚动一行
REQ_SCR_HBLINE	在域内水平向后滚动一行
REQ_SCR_HFHALF	在域内水平向前滚动半行
REQ_SCR_HBHALF	在域内水平向后滚动半行
REQ_VALIDATION	验证域
REQ_NEXT_CHOICE	显示下一个选择域
REQ_PREV_CHOICE	显示上一个选择域

在处理移动请求时，知道将要处理哪个域通常是很重要的。还有可能要设置窗体上的当前域。这两个函数能够完成上述功能：

```
FIELD *current_field(const FORM *form)
int set_current_field(FORM *form, FIELD *field)
```

判断一个域是否被改变也是很重要的。下面的例程能让你设置一个域的状态以及检查域的状态。非零状态意味着一个域被改变了：

```
int set_field_status(FIELD *field, bool status)
bool field_status(const FIELD *field)
```

还可以改变附加到一个窗体上的域。另外，窗体库支持页的概念，所以这些例程在和窗体相关联的域列表中的某些域上设置了页隔断。

```
int set_form_fields(FORM *form, FIELD **fields)
```

把窗体的域设置为新传递给窗体的域列表。这个列表必须以 NULL 结尾。

```
FIELD **form_fields(const FORM *form)
```

返回当前和参数 form 相关联的域数组。

```
int set_new_page(FIELD *field, bool new_page_flag)
```

如果 new_page_flag 为 TRUE，设置域从一个新页开始。FALSE 则取消 new_page_flag 设置。

```
int set_form_page(FORM *form, int n)
```

设置窗体的当前页。

```
int form_page(const FORM *form)
```

返回当前页号。

```
int field_index(const FIELD *field)
```

返回和窗体相关联的域数组中这个域的索引号。

```
bool new_page(const FIELD *field)
```

如果域标记了新页的开始则返回 TRUE，如果没有标记新页则返回 FALSE。

```
int field_count(const FORM *form)
```

返回当前和参数 form 相关联的域的数量。

```
int move_field(FIELD *field, int frow, int fcol)
```

把作为参数传递给该函数的域移动到屏幕的一个选定位置上。一定不要用在窗体身上。

```
int pos_form_cursor(const FORM *form)
```

当光标被窗体库以外的调用移动时，在当前选中域的光标的位置。

```
bool data_ahead(const FORM *form)
```

如果在作为参数传递给该函数的窗体前面有数据但又不在屏幕上时，返回 TRUE。

```
bool data_behind(const FORM *form)
```

如果在作为参数传递给该函数的窗体后面有数据但又不在屏幕上时，返回 TRUE。

另外，域可以被复制和链接。被复制的域是完整的副本，而被链接的域则共享缓冲。如果你要在不同的页上使用相同的域，这一功能就能派上用场。还有例程能改变域以及判断改动了什么。下面的例程完成上述功能：

```
FIELD *new_field(int height, int width, int toprow, int leftcol,
                 int offscreen, int nbuffers)
```

分配一个新域。必须给这个函数传递域的高度和宽度，在 toprow 和 leftcol 中提供域的左上角的位置，在 offscreen 中提供跨越屏幕的行数，并且提供额外的工作缓冲的数量。

```
FIELD *dup_field(FIELD *field, int toprow, int leftcol)
```

复制作为参数传递给该函数的域，而且只改变 toprow 和 leftcol。

```
FIELD *link_field(FIELD *field, int toprow, int leftcol)
```

链接到作为参数传递给该函数的域，而且只改变 toprow 和 leftcol。这两个域将共享它们的缓存。

```
int free_field(FIELD *field)
```

释放域使用的内存。

```
int set_max_field(FIELD *field, int max)
```

设置动态域的最大尺寸。

```
int field_info(const FIELD *field, int *rows, int *cols, int *frow,
               int *fcol, int *nrow, int *nbuf)
```

返回域在创建时设置的属性。

```
int dynamic_field_info(const FIELD *field, int *rows, int *cols,
                       int *max)
```

返回当前能改变的一个域的属性，也就是说当前的尺寸和最大尺寸。

正像菜单项一样，域也能够改变它的外观。默认设置可以在大多数终端下工作，但是对于更现代的终端来说，你还能加入彩色和加亮显示的域，给用户提供更多信息。

```
int set_field_fore(FIELD *field, chtype attr)
```

设置域的前景属性。默认为 A_STANDOUT。

```
chtype field_fore(const FIELD *field)
```

返回域的前景属性。

```
int set_field_back(FIELD *field, chtype attr)
```

设置域的背景属性。默认为 A_STANDOUT。

```
chtype field_back(const FIELD *field)
```

返回域的背景属性。

```
int set_field_pad(FIELD *field, int pad)
```

设置用来填充域的字符文本。默认为 A_STANDOUT。

```
int field_pad(const FIELD *field)
```

返回域的填充字符。

```
int set_field_just(FIELD *field, int justification)
```

设置域的调整值。可能的设置有 NO_JUSTIFICATION、JUSTIFY_RIGHT、JUSTIFY_LEFT 和 JUSTIFY_CENTER。

```
int field_just(const FIELD *field)
```

返回当前的调整设置。

每个域都至少有一个与之相关联的缓冲。窗体库只使用第一个缓冲，但是你能访问和控制其他缓冲。第一个例程设置当前缓冲的内容。第一个缓冲的编号为 0，第二为 1，依此类推。第二个例程返回缓冲的内容。另外，还有能够控制特定应用的数据；这些函数和菜单库中的函数类似，但用于窗体结构。

```
int set_field_buffer(FIELD *field, int buf, const char *value)
char *field_buffer(const FIELD *field, int buffer)
int set_field_userptr(FIELD *field, void *userptr)
```

```
void *field_userptr(const FIELD *field)
```

还有几个选项可以用来配置一个域。下面介绍能够控制它们的函数。表 24.4 介绍实用的选项。注意，当创建域时，所有的选项都设置为打开。

表 24.4 域配置选项

选项名称	描述
O_VISIBLE	显示域。如果这个选项被关闭，则禁止显示域
O_ACTIVE	域在处理过程中可以被访问。如果这个选项被关闭，不能通过移动键到达这个域。 请注意，一个不可见的域也同样是不活动的
O_PUBLIC	当输入数据时显示域的内容
O_EDIT	域可以被编辑
O_WRAP	不能在一行放下的单词被折到下一行显示。单词由空格隔开
O_BLANK	只要在域的第一个位置输入一个字符就清空整个域
O_AUTOSKIP	当一个域填满后就跳到下一个域
O_NULLOK	允许一个空白域
O_STATIC	域缓冲被修正到域最初的大小
O_PASSOK	只有在用户修改域时验证域

下面的例程和菜单库中设置菜单项选项的例程类似。对一个窗体的配置更多，所以这些选项也更有用：

```
int set_field_opts(FIELD *field, OPTIONS opts)
```

把作为参数传递给该函数的一个域的选项都设置为打开，其余选项设置为关闭。

```
int field_opts_on(FIELD *field, OPTIONS opts)
```

把作为参数传递给该函数的一个域的选项都设置为打开，并且保持其余选项不动。

```
int field_opts_off(FIELD *field, OPTIONS opts)
```

把作为参数传递给该函数的一个域的选项都设置为关闭，并且保持其余选项不动。

```
OPTIONS field_opts(const FIELD *field)
```

返回当前选项设置。

域类型用于验证数据。窗体库有几种内建的类型，但是下面描述的函数能够产生更多的类型。`set_field_type` 函数用来设置一个域的类型。在我们开始讨论怎样创建新的域类型并把它们附加到域上之前，先看看窗体库提供的域类型。

- **TYPE_ALPHA** 接受字母数据：没有空白、没有数字、没有特殊字符（这在输入字符时进行检查）。它的语法如下：

```
set_field_type(field, TYPE_ALPHA, width);
```

这里的 `width` 是一个 `int` 类型的整数，它设置了数据的最小宽度。把它设置为域的

宽度，或者如果它是一个可选域则设为 0。

- **TYPE_NUM** 接受字母数据和数字：没有空白、没有特殊字符（这在输入字符时进行检查）。它的语法如下：

```
set_field_type(field, TYPE_ALNUM, width);
```

这里的 `width` 是一个 `int` 类型的整数，它设置数据的最小宽度。把它设置为域的宽度，或者如果它是一个可选域则设为 0。

- **TYPE_ENUM** 这个类型让你把域限定在只能输入一个枚举类型的数据集合。它的语法如下：

```
field_type(field, TYPE_ENUM, valuelist, checkcase, checkunique);
```

这里的 `valuelist` 是一个以 `NULL` 结尾的字符串数组 (`char **`)，而 `checkcase` 是一个 `int` 类型的整数，如果它的值不为 0，则对用户输入和 `valuelist` 进行区分大小写的比较。`checkunique` 是一个 `int` 类型的整数，如果用户输入的字符串不能完全匹配，则迫使比较找出惟一的前缀。正常情况下，部分但不惟一的匹配会从 `valuelist` 中取出第一个匹配字符串。

- **TYPE_INTEGER** 这个域类型接受一个整数。它的语法如下：

```
set_field_type(FIELD *field, TYPE_INTEGER, padding, vmin, vmax);
```

有效字符由一个可选的前导负号后跟数字所组成。这些参数都是 `int` 类型的变量，而 `padding` 是给有效输入前面填充 0 的个数；0 意味着没有填充。参数 `vmin` 和 `vmax` 限制了有效输入的范围。

- **TYPE_NUMERIC** 这域类型接受一个十进制数。它的语法如下：

```
set_field_type(FIELD *field, TYPE_NUMERIC, padding, vmin, vmax);
```

有效字符由一个可选的前导负号后跟数字所组成，其小数部分也是数字。函数还接受一个小数点，但是它必须使用一个小数点的本地窗体。变量 `vmin` 和 `vmax` 是 `double` 类型的双精度变量，而 `padding` 是一个 `int` 类型的整数，它是给有效输入前面填充 0 的个数；0 意味着没有填充。参数 `vmin` 和 `vmax` 限制了有效输入的范围。

- **TYPE_REGEX** 这个域类型接受和一个正则表达式匹配的数据。它的语法如下：

```
set_field_type(FIELD *field, TYPE_REGEX, regexp);
```

参考 `regcomp` 的手册页面了解有关正则表达式的更多信息。

控制域类型的例程如下：

```
FIELDTYPE *new_fielddtype(bool (*const field_check) (FIELD *, const void *), bool (*const char_check) (int, const void *))
```

这个函数创建一个新的域类型。参数 `field_check` 必须返回 `TRUE` 或 `FALSE`，这取决于离开域后对域的确认；当有输入时 `char_check` 处理它。

```
int free_fielddtype(FIELDTYPE *fielddtype)
```

释放为 `fieldtype` 分配的内存。

```
int set_fieldtype_arg(FIELDTYPE *fieldtype,
                     void* (*const make_arg)(va_list *),
                     void* (*const copy_arg)(const void *),
                     void* (*const free_arg)(void *) )
```

这个函数设置 `fieldtype` 参数。

```
int set_fieldtype_choice(FIELDTYPE *fieldtype,
                        bool (*const next_choice)(FIELD *, const void *),
                        bool (*const prev_choice)(FIELD *, const void *) )
```

这个函数设置 `fieldtype_choice` 参数。

```
FIELDTYPE *link_fieldtype(FIELDTYPE *type1, FIELDTYPE *type2)
```

这个函数链接 `fieldtype`。

```
int set_field_type(FIELD *field, FIELDTYPE *type, ...)
```

这个函数设置窗体的域类型。

```
FIELDTYPE *field_type(const FIELD *field)
```

返回域类型。

```
void *field_arg(const FIELD *field)
```

返回传递给验证例程的参数。

```
int set_field_init(FORM *form, void (*func)(FORM *))
```

在修改域之后重画窗体时调用的函数指针。

```
void(*) (FORM *)field_init(const FORM *form)
```

返回 `field_init` 钩子函数的函数指针，如果没有设置它则返回 `NULL`。

```
int set_field_term(FORM *form, void (*func)(FORM *))
```

在修改域之前未重画窗体时调用的函数指针。

```
void(*) (FORM *)field_term(const FORM *form)
```

返回 `field_term` 钩子函数的函数指针，如果没有设置它则返回 `NULL`。

```
int set_form_init(FORM *form, void(*func)(FORM *))
```

修改页之后重画窗体时调用的函数指针。

```
void(*) (FORM *)form_init(const FORM *form)
```

返回 `form_init` 钩子函数的函数指针，如果没有设置它则返回 `NULL`。

```
int set_form_term(FORM *form, void(*func)(FORM *))
```

修改页之前未重画窗体时调用的函数指针。

```
void(*) (FORM*) form_term(const FORM *form)
```

返回 form_term 钩子函数的函数指针，如果没有设置它则返回 NULL。

24.4.3 示例程序

程序清单 24.3 是一个基本的窗体，它包含了两个默认域，要求输入用户的名和姓。最后一个域模仿了一个按钮——在其中按回车键会退出程序。

程序清单 24.3 chap24.3.c

```
/*
 * Listing 24.3
 */
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <ctype.h>
#include <curses.h>
#include <form.h>
#include <errno.h>
#include "utilfcns.h"

int
main(int argc, char *argv[])
{
    int quit = 0, key;
    FORM *form;
    FIELD *fields[5];

    app_init();

    fields[0] = new_field(1, 20, 2, 2, 0, 0);
    field_opts_off(fields[0], O_ACTIVE);
    set_field_buffer(fields[0], 0, "First Name:");
    fields[1] = new_field(1, 50, 3, 5, 0, 0);
    fields[2] = new_field(1, 20, 4, 2, 0, 0);
    field_opts_off(fields[2], O_ACTIVE);
    set_field_buffer(fields[2], 0, "First Name:");
    fields[3] = new_field(1, 50, 5, 5, 0, 0);
    fields[4] = new_field(1, 10, 6, 5, 0, 0);
    set_field_buffer(fields[4], 0, "--QUIT--");
    field_opts_off(fields[4], O_EDIT);
    fields[5] = NULL;
    form = new_form(fields);
    post_form(form);
    raw();
    noecho();
    keypad(stdscr, TRUE);
    while (!quit) {
```

```
int code;

key = getch();
switch (key) {
    case KEY_DOWN:
    case KEY_RIGHT:
    case '\t':
        code = form_driver(form, REQ_NEXT_FIELD);
        break;
    case '\n':
        if (field_index(current_field(form)) == 4) {
            quit = 1;
        } else {
            code = form_driver(form, REQ_NEXT_FIELD);
        }
        break;
    case KEY_UP:
    case KEY_LEFT:
        code = form_driver(form, REQ_PREV_FIELD);
        break;
    default:
        if (isprint(key)) {
            form_driver(form, key);
        } /* else ignore it */
        break;
}

app_exit();
exit(0);
}
```

24.5 小 结

ncurses 提供的高级特性能够用来创建现代风格的用户界面。在这一章里，我们介绍了能够使用鼠标的终端和使用这种类型终端的 ncurses 接口。我们介绍了能够创建用户界面的菜单接口，用 Menu API 创建的用户界面可以向用户提供一种简单的可选集合，Menu API 能够处理几乎所有的屏幕更新和用户输入。最后，我们介绍了 Form API，它能让用户创建复杂的窗体。窗体和菜单库特别需要花时间学习，在开发易于使用的基于终端的应用过程中，它们极大地减少了代码量。

第 25 章 X Windows 编程

X Windows 系统是在大多数 UNIX 系统上使用的图形用户界面。它是基于网络的 GUI 系统，并采用了一种客户机/服务器的概念。在 X Windows 系统中运行的应用都被称为客户 (client)。客户程序并不直接在屏幕上绘制或操纵任何图形；而是和 X 服务器进行通信。服务器完成所有的绘图工作并且控制有关显示的各个方面。因为在客户程序和服务器间的所有通信都基于网络进行的，这就意味着你可以在远程的计算机上运行程序而在本地的屏幕上显示它的 GUI。

X Windows 系统最初是在麻省理工学院 (MIT) 作为 Athena 计划的一部分开发出来的。今天的 X Windows 系统由作为 Open Group 一部分的 X Consortium 维护。这个联盟在 20 世纪 80 年代中期发布了 X Windows 系统的第一个版本。在那时，“开放 (open)” 的含义和今天的并不相同。开放通常意味着如果你愿意支付一笔不菲的费用，签订一份保密协议而且保证你的后代也过上奴隶般的生活，就可以得到规范说明——当然最后一点形容得过于夸张。

如果你正在使用 Linux，那么就有可能使用 X Windows 系统的 XFree86 版本。这个版本是 X Windows 系统的免费重新发布的、开放源代码的实现。它是由一个非盈利性的组织 XFree86 Project Inc 开发的。这是一个非常出色的版本，而且无论是当作一个用户环境还是当作一个开发环境，它都能满足你的各种需要。Linux 上也有商业版本的 X Windows 系统。

X Windows 的网络方面体现了一个非常强大的概念，但是和所有的好东西一样，它也伴随着一定的代价。这里的代价就是由于客户端和服务端间的网络通信带来的开销造成速度上的不足。但是，客户机/服务器解决方案带来的可能性足以弥补上述不足。例如，你可以在另一个计算能力更强的计算机上运行你的应用程序，而在本地计算机上显示图形用户界面。即使正在运行的计算机所用的体系结构与你的不同，也能做到这一点。因为 X Windows 使用标准的 TCP/IP 连接传送网络流量，你可以在任何计算机上使用你的应用，而不必考虑它的地理位置，只要你能通过网络访问它即可。作为一个程序员，你不必关心 X Windows 的连网功能如何工作，因为它们对程序员和用户来说都是透明的。

在本章中，你将学会怎样使用原始的 X 库 (Xlib) 和 X 工具包 (Xt)。这些都是应该了解的好东西，但是应该牢记，如果你是一位应用软件开发人员，可能你只会用高层的 API 编写 X Windows 程序——比如我们将在下面几章介绍的那些 API。在第 26 章“Athena、Motif 和 LessTif 窗口部件”中，你将使用 MIT Athena 窗口部件集和 Motif 窗口部件集，两者都使用了 X 工具包。在第 27 章“使用 GTK+ 进行 GUI 编程”中，我们将介绍 GTK+ 窗口部件库。最后，在第 28 章“使用 Qt 进行 GUI 编程”中，我们将介绍 Qt 窗口部件库。

注意： X Windows 通常被认为是用于 UNIX 系统的窗口系统，但是 X 服务器也能运行在 OS/2、Windows 和 Macintosh 操作系统上。

25.1 X 的概念

正如在介绍中所说明的那样，X Windows 基于一种客户机/服务器的思想。它将显示和事件处理功能从应用程序（或者通常称作客户的程序）中分离出来。相反，一个客户端应用程序通过套接口接口和 X 服务器进行通信——这种通信对用户透明，而且既可以是本地的也可以通过网络进行。

因为 X Windows 是事件驱动的，它会花费大量自己的时间处于一种等待事件发生的状态。X 服务器处理所有的 I/O 资源，比如键盘输入、鼠标输入以及显示屏幕。一旦这些资源产生了事件，服务器会根据需要把它们返回给客户程序。例如，当用户单击鼠标时，X 服务器检测到鼠标事件出现的位置并且把鼠标事件发送给适当的客户程序。另一个例子是当显示器上原先被其他窗口遮住的窗口变成可见时，X 服务器将发送给相应的应用程序一个窗口暴露事件。当一个窗口的部分或全部变成可见之后就发生窗口暴露事件，因此需要重画窗口。客户程序的响应通常是向 X 服务器发回绘图操作，指示它如何重画窗口内容。因为 X Windows 是无状态的，除非客户程序重画被暴露的部分，否则一个窗口被暴露部分的内容会成为空白。

为了理解 X Windows 应用程序和 X 服务器之间的交互关系，你需要理解事件是怎样被处理的，并且掌握一个应用程序可以请求 X 服务器进行的绘制操作。图 25.1 显示了实际的用户事件、X 服务器的事件队列以及应用程序事件队列之间的相互关系。

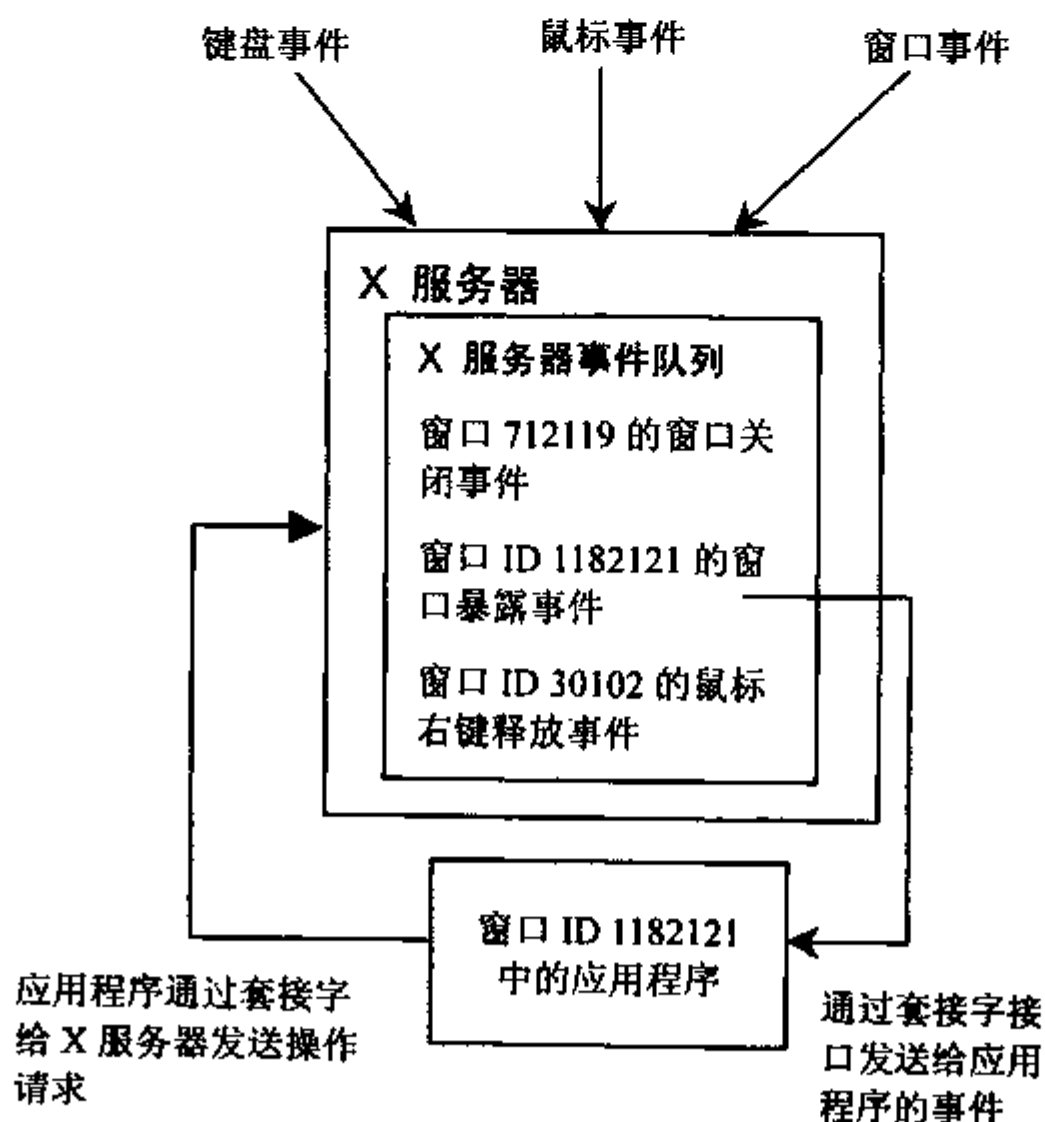


图 25.1 事件、X 服务器以及应用程序之间的交互

客户机和服务器之间的通信通过一个称为 Xlib 的低层接口来执行。这是 X Windows 软件体系结构中的最低层。你可以使用它来编写应用程序。但是，这样做非常花时间，和自己重新发明车轮差不多。

为了简化 X Windows 编程，在 Xlib 之上有好几层其他库。这些库通常称为工具包 (toolkit)。这些工具包中最重要的也是最低层的是 X 工具包内部函数 (X toolkit intrinsics)，即 Xt，这也是它最常见的称呼。它提供了构造窗口部件所需的基础，窗口部件是任何图形用户界面系统的基本组成单位。窗口部件的例子有按钮、菜单、滑动条等等。在大多数情况下，你都有至少一种在 X 工具包之上的其他级别的工具包。

正如图 25.2 中所显示的那样，X Windows 应用程序可以使用低层的 Xlib API、X 工具包内部函数 (或者 X intrinsics)、Athena 窗口部件集以及 Motif 窗口部件集的任意组合。在第 27 章和第 28 章中，你也会看到两种高层的 API：GTK+ 和 Qt 图形用户界面库。

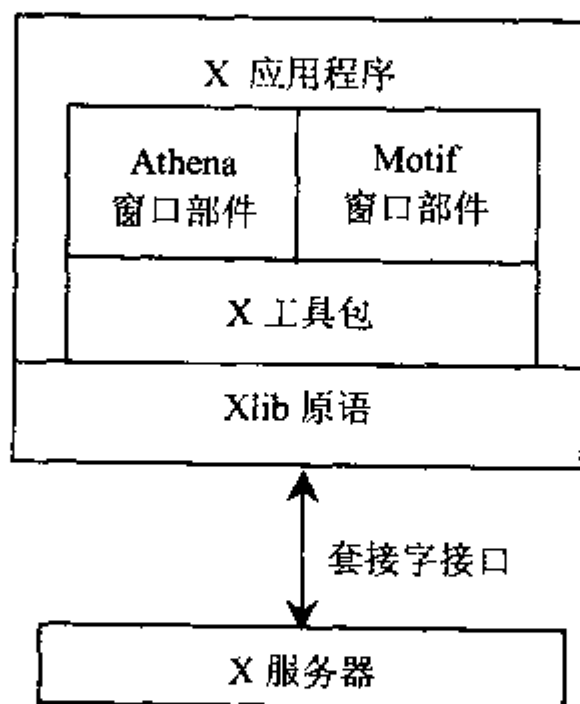


图 25.2 X Windows 编程 API

25.2 Xlib API

对于大多数要开发的 X Windows 应用程序来说，你可能需要使用一些高层工具包，比如 Motif、Athena、GTK+ 或者 Qt。但是，除了在高层工具包和 X 服务器之间提供软件接口之外，Xlib API 还提供了在面向图形的应用中可能用到的有用的图形操作。位于 `src/X/xlib` 目录下的示例程序 `bifurcation.c` 提供了一个简单的创建窗口、处理事件以及使用 Xlib API 所提供的简单图形操作的例子。

提示： Christophe Tronche 在他的 Web 网站 <http://www.tronche.com/gui/x/> 上维护了一个出色的 Xlib 应用程序编程接口参考。

你的 X Windows 程序建立并显示一个窗口需要经过几个基本步骤。本节将更深入地介

绍在这期间涉及到的所有 API 调用。

1. 首先需要打开一个到 X 服务器的连接。这可以用 API 调用 `XOpenDisplay` 完成。
2. 然后使用 `DefaultScreen` 获得一个指向默认屏幕的指针。
3. 现在为自己的应用创建一个窗口。我们既可以用 `XCreateWindow`，也可以用 `XCreateSimpleWindow`。两者都完成了同样的功能，但是后者使用了一些默认值，所以参数较少。
4. 需要指定 Xlib 应该告诉客户的事件。这通过 `XSelectInput` 完成。
5. 最后，必须在 X Windows 系统中显示窗口，这通过 `XMapWindow` 完成。

在开始查看本章的示例程序之前，先考察一下 Xlib 在管理显示器和窗口方面的功能，列出常见的事件处理函数，并且查看一些常用的图形原语。这些函数都建立在 Xlib 和 Xt 函数的基础上。这些函数中间有些是纯粹的废物，它们只是作为遗迹被保存下来。常用函数的数量相当少，完全能够管理。下面的各节介绍了最常用的窗口和显示器管理函数。这些函数按照它们在 X 应用中常用的顺序而不是字母顺序列举出来的。

25.2.1 XOpenDisplay

任何 X 程序需要做的第一件工作就是使用 `XOpenDisplay` 连接到 X 服务器。你可以提供一个显示器名作为这个调用的参数。这个函数还能用来在一个特定计算机或 X 屏幕上打开显示器。显示器名具有通用格式，`hostname:display-number:screen-number`。你可能只想简单地使用 `localhost` 以及默认的显示器和屏幕数量。你也能发送一个 `NULL` 指针作为显示器名，此时程序的窗口将在用户默认的 X 显示器上打开。这可能是这个函数最常见的用法。

该函数的原型如下：

```
Display *XOpenDisplay(char *display_name)
```

25.2.2 XCreateSimpleWindow 和 XCreateWindow

这些 API 调用是用来创建窗口的。用这些函数创建的窗口直到使用 `Xmap Window` 函数命令它们之后才会显示出来。函数 `XCreateWindow` 是在 X 显示器上创建新窗口的通用函数。一个更简单的函数 `XCreateSimpleWindow` 则使用从其父窗口继承的默认值来创建窗口。后一个版本可能最为常用，因为它的参数更少。这两个函数的特征如下：

```
Window XCreateWindow(Display *display, Window parent,
                    int x, int y,
                    int width, int height,
                    unsigned int border_width,
                    unsigned int depth, int class,
                    Visual * visual,
                    unsigned long valuemask,
                    XSetWindowAttributes * attributes)
Window XCreateSimpleWindow(Display * display,
                          Window parent,
                          int x, int y,
                          unsigned int width,
```

```

        unsigned int height,
        unsigned int border_width,
        unsigned long border,
        unsigned long background)

```

你能调整 n 个参数来改变一个窗口的外观。这可以用 `XsetWindowAttributes` 结构来完成 `XSetWindowAttributes` 结构定义如下：

```

typedef struct {
    Pixmap background_pixmap;           // background
    unsigned long background_pixel;      // background pixel
    Pixmap border_pixmap;               // border of the window
    unsigned long border_pixel;          // border pixel value
    int bit_gravity;                    // one of bit gravity values
    int win_gravity;                    // one of the window gravity values
    int backing_store;                  // NotUseful, WhenMapped, Always
    unsigned long backing_planes;        // planes preserved
                                         // if possible
    unsigned long backing_pixel;         // value for restoring planes
    Bool save_under;                    // should bits under be saved?
    long event_mask;                    // set of saved events
    long do_not_propagate_mask;          // set of events that should
                                         // not propagate
    Bool override_redirect;              //boolean value for
                                         // override_redirect
    Colormap colormap;                  // color map to be associated
                                         // with window
    Cursor cursor;                       // cursor to be displayed (or None)
}XSetWindowAttributes;

```

25.2.3 映射窗口和撤销映射窗口

如前所述，一个 X 窗口启动时并不会在屏幕上显示出来。为了让它能看得见，你需要用 `XmapWindow`。如果你愿意，你可以随时让窗口隐藏或显示。X Window 是通过映射操作变为可见的，而通过撤销映射变为不可见。用来控制窗口可见性的实用函数原型定义如下所示：

```

XMapWindow(Display *display, Window w)
XMapSubwindows(Display * display, Window w)
XUnmapWindow(Display * display, Window w)

```

窗口在映射后可能并不会立刻变为可见的，这是因为一个窗口要变为可见的，这个窗口以及它的所有父窗口必须都经过映射。这种操作非常方便，因为你只要撤销最上层窗口的映射，就可以使整个嵌套（或子窗口）树上的窗口变为不可见。当你撤销一个窗口的映射时，X 服务器自动为每一个重新可见的窗口产生一个窗口暴露事件。

25.2.4 撤销窗口

一旦窗口不再显示，就调用非映射函数撤销它，而不是隐藏该窗口，撤销该窗口可以释放系统资源。函数 `XDestroyWindow` 撤销单独一个窗口资源，而 `XDestroySubWindows` 用来撤销所有的子窗口资源。这些函数的原型定义如下所示：

```
XDestroyWindow(Display * display, Window w)
XDestroySubwindows(Display * display, Window w)
```

25.2.5 事件处理

使用 Xlib API 进行事件处理并不是非常困难。一个程序必须使用 `XSelectInput` 对它感兴趣的事件类型进行登记，然后使用 `XNextEvent` 来检查从 X 服务器发出的事件。

XSelectInput

这个 API 调用用来登记 X Window 将向你的程序报告哪些事件。默认情况下，不报告任何事件，你需要特意打开你的程序可能感兴趣的每一个事件。`XSelectInput` 的函数原型定义如下：

```
XSelectInput(Display * display, Window w, long event_mask)
```

变量 `display` 定义了应用程序使用哪个 X 服务器用于显示。事件是指定的窗口，并通过事件标志和事件掩码进行组合来设置。表 25.1 列出了最常用的事件标志。

表 25.1 最常用的事件标志

事件标志	说明
<code>KeyPressMask</code>	需要键盘按下事件
<code>KeyReleaseMask</code>	需要键盘释放事件
<code>ButtonPressMask</code>	需要鼠标按钮按下事件
<code>ButtonReleaseMask</code>	需要鼠标按钮释放事件
<code>EnterWindowMask</code>	需要鼠标进入窗口事件
<code>LeaveWindowMask</code>	需要鼠标离开窗口事件
<code>PointerMotionMask</code>	需要鼠标移动事件
<code>PointerMotionHintMask</code>	需要鼠标移动提示事件
<code>Button1MotionMask</code>	需要在鼠标按钮 1 按下的同时鼠标指针移动事件
<code>Button2MotionMask</code>	需要在鼠标按钮 2 按下的同时鼠标指针移动事件
<code>Button3MotionMask</code>	需要在鼠标按钮 3 按下的同时鼠标指针移动事件
<code>ButtonMotionMask</code>	需要在鼠标任意按钮按下的同时鼠标指针移动事件
<code>ExposureMask</code>	需要暴露事件
<code>VisibilityChangeMask</code>	需要任何改变可见性的事件
<code>ResizeRedirectMask</code>	需要重定向窗口大小改变事件
<code>FocusChangeMask</code>	需要输入焦点改变事件

为了获得更多的 Xlib 文档，可以查询 Christophe Tronche 的 Web 站点或者 X 协会在 www.x.org 上的网站。

XNextEvent

一旦你告诉了 X Windows 你的程序能处理什么类型的事件, 你就需要检查事件的发生, 并且检索与它们有关的数据。应用程序可以使用函数 `XNextEvent` 来获得下一个被挂起的事件。该函数的原型定义如下所示:

```
XNextEvent(Display * display, XEvent * event_return_value)
```

`XEvent` 结构具有一个域类型, 用来指明发生了什么事件。这里, 你应该检查 `value` 结构的 `type` 成员的内容, 即 `value→type`。表 25.2 列出了最常用的事件类型。

表 25.2 最常用的事件类型

事件类型	说明
<code>ButtonPressMask</code>	按下任何鼠标按钮
<code>Button1PressMask</code>	按下鼠标按钮 1
<code>Button2PressMask</code>	按下鼠标按钮 2
<code>Button3PressMask</code>	按下鼠标按钮 3
<code>ButtonMotionMask</code>	按下任何按钮的同时鼠标移动
<code>Button1MotionMask</code>	按下按钮 1 的同时鼠标移动
<code>Button2MotionMask</code>	按下按钮 2 的同时鼠标移动
<code>Button3MotionMask</code>	按下按钮 3 的同时鼠标移动
<code>KeyPress</code>	按下键盘上任意键
<code>KeyRelease</code>	释放键盘上任意键
<code>Expose</code>	窗口被暴露 (大多数应用程序进行重画)

还有许多其他可能的事件类型, 但以上是最常用的。

25.2.6 初始化图形设备上下文和字体

X 中的每个窗口都有许多不同的设置。不同的绘制操作也在同一个窗口中使用不同的设置。例如, 不同的字体、背景和前景色以及多种其他绘制参数。为了在图形操作应该使用的设置上保留一个标记, 你需要给它提供一个图形设备上下文 (Graphics Context, GC)。

你可以从某些窗口部件得到一个指向 GC 的指针, 也可以调用一次 `XcreateGC` 函数创建一个新的指针。该函数的声明如下:

```
GC XCreateGC(Display * display, Drawable d,
             unsigned long valuemask,
             XGCValues * values)
```

一个 `Drawable` 通常是一个 `Window` 对象, 但它也可以是个 `Pixmap`。

在定义了一个 GC 之后, 程序可以使用以下函数设置字体、前景色和背景色。

```
XSetFont(Display * display, GC gc, Font font)
XSetForeground(Display * display, GC gc, unsigned long a_color)
XSetBackground(Display * display, GC gc, unsigned long a_color)
```

一个 `XfontInfo` 数据结构是用 `XloadFont` 定义的，它的函数原型如下：

```
XfontStruct *XloadQueryFont(Display * display, char * name)
```

示例字体的名称为 `8×10`、`9×15` 和固定。

一个颜色值可以用 `XParseColor` 或者以下宏之一获得：

```
BlackPixel(Display * display, Screen screen)
WhitePixel(Display * display, Screen screen)
```

25.2.7 在 X 窗口中绘图

通常，在一个 X Window 中进行绘制是在程序接收到一个暴露事件之后进行的。在任何改变前景色以及其他选项的绘制命令发出之前，GC 仍可以改变。下面的绘制函数列表给出了一些 X Window 程序员可以使用的图形操作示例。

```
XDrawString(Display *display, Drawable d, GC gc,
            int x,int y,
            char * string, int string_length)
```

`XDrawString` 使用 GC 的当前字体和前景色绘制一个字符串。

```
XDrawLine(Display *display, Drawable d, GC gc,
           int x1, int y1, int x2, int y2)
```

`XDrawLine` 被用来在 $(x1,y1)$ 和 $(x2,y2)$ 两点之间绘制一条线段。

```
XDrawRectangle(Display *display, Drawable d, GC gc, int x,
               int y, unsigned int width, unsigned int height)
```

`XDrawRectangle` 使用 GC 当前的前景色绘制一个矩形。

```
XDrawArc(Display *display, Drawable d, GC gc, int x, int y,
          unsigned int width, unsigned int height,
          int angle1, int angle2)
```

这里，`angle1` 用来指 64 乘以单位弧度计算出的圆弧起始位置相对于三点整的时针位置（x 轴正半轴）的角度。参数 `angle2` 用来指圆弧相对于 `angle1` 的角度，同样是 64 乘以单位弧度计算出的。函数 `XDrawRectangle` 和 `XDrawArc` 以 GC 的前景色绘制物体形状的轮廓。函数 `XFillRectangle`（与 `XDrawRectangle` 有相同的参数）和 `XfillArc`（与 `XDrawArc` 有相同的参数）进行同样的操作，只是它们使用前景色填充物体轮廓。

25.2.8 一个 Xlib 的示例程序

在 `src/X/xlib` 目录下的源文件 `bifurcation.c` 显示了如何创建 X 窗口，如何俘获按钮按下事件，如何暴露和缩放事件，以及如何进行一些简单的图形操作。`bifurcation.c` 程序使用一个简单的由生物学家 Robert Mays 提出的非线性方程来绘制无秩序的人口增长模式。图 25.3 显示了 `bifurcation.c` 程序运行的情况。

这个简单的程序使用函数 `draw_bifurcation` 来绘制窗口内容。该函数的原型定义如下所示：


```
void draw_bifurcation(Window win, GC gc, Display *display,
                    int screen, Xcolor text_color
```

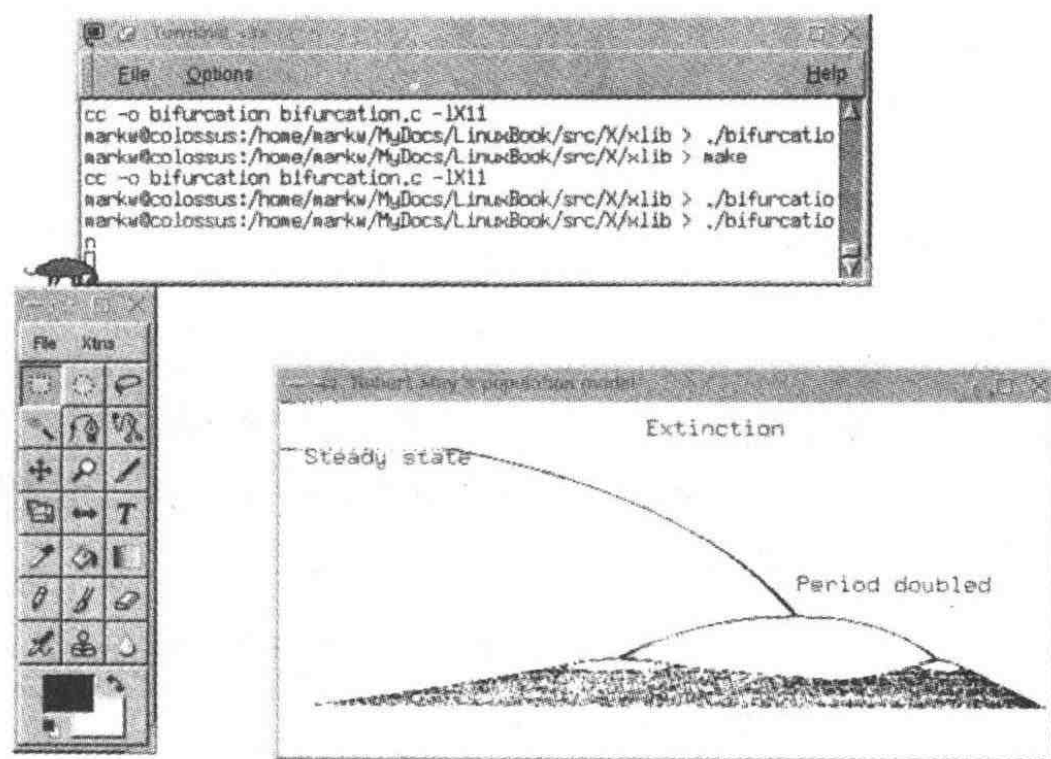


图 25.3 bifurcation.c 示例 Xlib 程序

函数 main 定义了下面的数据：

Display *display——用来指 X 服务器的显示器

int screen——用来确定你正在使用 X server 上的哪一个屏幕

Window win——用来确定应用程序窗口

XColor blue——你想要使用蓝颜色来显示文本

unsigned int width=500——指定起始窗口的宽度

unsigned int height=231——指定起始窗口的高度

XFontStruct *font_info——用来使用一种显示字体

GC gc——用来标识窗口的图形设备上下文 (GC)

Colormap cmap——用蓝色查询 X 服务器

XEvent x_event——用来获取 X 事件

XGCValues values——用来设置由 X 服务器返回的 GC 属性值

X Window 程序需要做的第一件事是确保它能够在需要的 X 服务器上打开一个显示器。

```
if((display=XopenDisplay(NULL)) == NULL ) {
    perror("Could not open X display");
    exit(1);
}
```

这里，如果没有可用的 X 显示器，程序将终止。将 XOpenDisplay 的参数设置为 NULL，将默认返回环境变量 DISPLAY 的值。你同样想使用本地 X 服务器的默认屏幕并创建窗口：

```
screen = DefaultScreen(display);
win = XcreateSimpleWindow(display, RootWindow(display, screen),
```

```
0,0,width,height,5,
BlackPixel(display,screen),
WhitePixel(display,screen));
```

因为想要蓝色文本，你需要调用下面的代码：

```
cmap=DefaultColormap(display,screen);
XParseColor(display,cmap,"blue",&blue);
XAllocColor(display,cmap,&blue);
```

如果请求了一个并不可用的颜色值，很可能你将获得一个随机颜色值，但程序不会因此而异常终止。作为一个小实验，可以试一试下面的代码：

```
XParseColor(display, cmap, "ZZ_NO_SUCH_COLOR",&blue);
```

不同于选择颜色值，在指定显示字体时发生的一个错误将导致 X 程序的异常终止。在下面的程序代码中，你将尝试两种几乎任何 X 服务器上都有字体：

```
if ((font_info = XLoadQueryFont(display, "9x15")) == NULL) {
    perror("Use fixed font\n");
    font_info = XLoadQueryFont(display, "fixed");
}
```

这里，如果你不能在 X 服务器上找到 9×15，将接下来尝试 fixed 字体。现在，你可以准备为窗口中的绘制操作创建一个图形设备上下文（GC）。

```
gc = XCreateGC(display,win,(unsigned long)0,&values);
XSetFont(display, gc, font_info>fid);
XSetForeground(display, gc, BlackPixel(display, screen));
```

在这里也可以为 GC 设置字体的前景色。

在映射窗口之前（使窗口变为可见的），你需要设置属性值以供窗口管理器使用，并选择你要处理的事件：

```
XSetStandardProperties(display,win,
    "Robert May's population model",
    "Bifurcation",None,
    0, 0, NULL);
XSelectInput(display, win, ExposureMask | ButtonPressMask);
```

到目前为止，你所使用的都是默认值，除了窗口标题和在应用程序被图标化时由窗口管理器使用的标签。对 XSelectInput 的调用将通知 X 服务器你想要接收暴露事件和鼠标按钮按下事件。最后，你准备使窗口变为可见，并绘制窗口内容：

```
XMapWindow(display,win);
draw_bifurcation(win,gc,display,screen,blue);
```

在函数 main 中仅剩的要做的事就是处理事件：

```
while (1) {
    XNextEvent(display, &x_event);
```

```

switch (x_event.type) {
case Expose:
    draw_bifurcation(win,gc,display,screen,blue);
    break;
case ButtonPressMask:
    XCloseDisplay(display);
    exit(0);
default:
    break;
}
}

```

上面的事件循环很简单：你调用 `XnextEvent`（该函数在调用后将一直处于阻塞状态，直到有事件发生为止）来填写 `XEvent x_event` 变量。`X_event.type` 域是一个整数事件类型，用来和事件常量 `Expose` 以及 `ButtonPressMask` 进行比较。如果获得一个暴露事件，窗口将被重画，因此你要调用 `draw_bifurcation` 函数。如果使用者在窗口中按下了任何鼠标按钮，你将使用 `XCloseDisplay` 来关闭与 X 服务器的连接并终止程序。

函数 `draw_bifurcation` 相当简单：

```

void draw_bifurcation(Window win, GC gc, Display *display,
int screen, Xcolor font_Color) {
    float lambda = 0.1;
    float x = 0.1f;
    float population = 0.0;
    int x_axis, y_axis, iter;
    XSetForeground(display, gc, font_color.pixel);
    XDrawString(display, win, gc, 236,22,
        "Extinction",strlen("Extinction"));
    XDrawString(display,win, gc, 16,41,
        "Steady state",strlen("Steady state"));
    XDrawString(display, win,gc,334,123,
        "Period doubled",strlen("Period doubled"));
    XSetForeground(display,gc,BlackPixel(display,screen));
    for (y_axis=0; y_axis<198; y_axis++) {
        lambda = 4.0f * (0.20f + (y_axis / 250.0f));
        for(iter=0; iter<198; iter++) {
            population = lambda * x * (1.0f - x);
            x_axis = (int)(population * 500.02f);
            if (x_axis > 0.0f && x_axis < 501.0f) {
                XDrawLine(display,win,gc,x_axis,y_axis,x_axis,y_axis);
            }
            x = population;
        }
    }
}

```

鉴于本书的目的，决定在 X 窗口的什么位置画点的算法并不是我们所感兴趣的。下面的函数是用来在窗口的图形设备上下文中进行绘制操作：

XSetForeground——用来在黑色和蓝色之间切换前景色

XDrawString——用来在窗口中使用 GC 中设置的字体写三个文本串

XDrawLine——用来在窗口中画点

你调用 XSetForeground 来改变写文本和绘制直线的颜色值。这里，你是通过每点一个像素长的画点操作来绘制直线段的。下面，进入目录 src/X/xlib，然后键入下面的命令来运行示例程序 bifurcation.c。

```
make
./bifurcation
```

这个例子的完整源代码见程序清单 25.1。

程序清单 25.1 bifurcation.c

```
// bifurcation.c
//
// Copyright Mark Watson, 1999. Open Source Software License
//
// This program implements biologist Robert May's
// population growth model. The output graphic shows
// regimes of stability and chaos in population size.
//

#include <stdio.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>

void draw_bifurcation (Window window, GC gc, Display *display,
                      int screen, XColor text_color);

main(int argc, char **argv) {
    Display *display;
    int screen;
    Window win;
    XColor blue;
    unsigned int width=500, height=231;
    XFontStruct *font_info;
    GC gc;
    Colormap cmap;
    XEvent x_event;
    XGCValues values;

    if ( (display=XOpenDisplay(NULL)) == NULL ) {
        perror("Could not open X display");
        exit(1);
    }
}
```

```

screen = DefaultScreen(display);

win = XCreateSimpleWindow(display, RootWindow(display, screen),
                          0, 0, width, height, 5,
                          BlackPixel(display, screen), // border
                          WhitePixel(display, screen)); // background

cmap=DefaultColormap(display,screen);

XParseColor(display,cmap,"blue",&blue);
XAllocColor(display,cmap,&blue);

if ((font_info = XLoadQueryFont(display,"9x15")) == NULL) {
    perror("Use fixed font\n");
    font_info = XLoadQueryFont(display,"fixed");
}

gc = XCreateGC(display,win,(unsigned long)0,&values);

XSetFont(display, gc, font_info->fid);
XSetForeground(display, gc, BlackPixel(display,screen));

XSetStandardProperties(display, win,
                      "Robert May's population model",
                      "Bifurcation", None,
                      0, 0, NULL);

XSelectInput(display, win, ExposureMask | ButtonPressMask);
XMapWindow(display, win);
draw_bifurcation(win,gc,display,screen,blue);

while (1) {
    XNextEvent(display, &x_event);
    switch (x_event.type) {
        case Expose:
            draw_bifurcation(win,gc,display,screen,blue);
            break;
        case ButtonPressMask:
            XCloseDisplay(display);
            exit(0);
        default:
            break;
    }
}

void draw_bifurcation(Window win, GC gc, Display *display,
                      int screen, XColor font_color) {

    float lambda = 0.1;
    float x = 0.1f;
    float population = 0.0;

```

```

int x_axis, y_axis, iter;

XSetForeground(display, gc, font_color.pixel);
XDrawString(display, win, gc, 236, 22,
             "Extinction", strlen("Extinction"));
XDrawString (display, win, gc, 16, 41,
             "Steady state", strlen("Steady state"));
XDrawString (display, win, gc, 334, 123,
             "Period doubled", strlen("Period doubled"));

XSetForeground(display, gc, BlackPixel(display, screen));
for (y_axis=0; y_axis<198; y_axis++) {
    lambda = 4.0f * (0.20f + (y_axis / 250.0f));
    for (iter=0; iter<198; iter++) {
        population = lambda * x * (1.0f - x);
        x_axis = (int)(population * 500.02f);
        if (x_axis > 0.0f && x_axis < 501.0f) {
            XDrawLine(display, win, gc, x_axis, y_axis, x_axis,
                      y_axis);
        }
        x = population;
    }
}
}

```

25.3 X Toolkit API

你已经看到 Xlib API 是编写 X Windows 应用的一种低层但非常有效的编程库。X 工具包（又称为内部函数）库提供了编写窗口部件的高层编程支持。

窗口部件是面向对象的显示对象，比如数据项域、绘制数据的工具等等。窗口部件通常是用 C 语言编写的，但从它们支持继承、维持私有数据，并提供一个访问窗口部件内部数据的公共 API 的方式来看，它们是面向对象的。在下一章中，你会使用两种最流行的窗口部件集：Athena 窗口部件集和 Motif 窗口部件集。

25.3.1 X Toolkit 使用入门

在一个应用程序可以使用 X 工具包之前，必须在任何其他工具包函数（类型 String 被定义为 char*，Cardinal 被定义为 int）之前调用下面的这个函数：

```

Widget XtInitialize(String shell_name,
                    String application_class,
                    XrmOptionDescRec *options,
                    Cardinal num_options,
                    int * argc, char **argv)

```

参数 `name` 和 `class` 指定了应用程序顶层窗口部件的名字和类别。典型的做法是，你可以为应用程序起一个有意义的名字，然后通过将应用程序名的第一个字母大写来创建一个类别名。参数 `options` 通常赋值为 `NULL` 即可，表示没有什么特别的选项。如果你使用 `NULL` 来指定 `options`，需要将 `num_options` 的值赋为 0。最后的两个参数是用来向应用程序函数 `main` 传递的参数（通过移去任何与 X 相关的参数获得）。

X 工具包使用由字符串来指定的资源。为了使程序的可读性更好，X 工具包和 Xlib 包含了那些定义资源字符串常量名的文件：

```
#define XtNwidth "width"
#define XtNheight "height"
#define XtNlabel "label"
```

使用窗口部件来编写 X 应用程序相当简单，但编写新的窗口部件却困难得多。在下面的两章中，我们将学习如何使用 Athena 和 Motif 窗口部件以及如何编写一个新的 Athena 窗口部件。

25.3.2 使用 X 工具包设置窗口部件参数

X 工具包函数 `XtSetValues` 可以被用来为窗口部件资源设置各种属性值。`XtSetValues` 既可用于 Athena 部件，也可用于 Motif 部件。每种类型的部件（标签、文本编辑器等）具有不同的可以通过 `XtSetValues` 进行设置的选项。`XtSetValues` 从资源内核（或基本部件类别）开始，然后沿着部件之间的继承链，从一个部件到另一个部件，设置那些与指定的资源名称相匹配的资源。例如，假设你正在使用一个 Athena 标签部件，对文件 `Label.h`（位于我的 Linux 系统上的 `/usr/X11R6/include/X11/Xaw3D` 目录下）的检查显示了标签部件所拥有的资源：

名字	类别	RepType	默认值
background	Background	Pixel	XtDefaultBackground
bitmap	Pixmap	Pixmap	None
border	BorderColor	Pixel	XtDefaultForeground
borderWidth	BorderWidth	Dimension	1
cursor	Cursor	Cursor	None
cursorName	Cursor	String	NULL
destroyCallback	Callback	XtCallbackList	NULL
encoding	Encoding	unsigned char	XawTextEncoding8bit
font	Font	XFontStruct*	XtDefaultFont
foreground	Foreground	Pixel	XtDefaultForeground
height	Height	Dimension	text height
insensitiveBorder	Insensitive	Pixmap	Gray
internalHeight	Height	Dimension	2
internalWidth	Width	Dimension	4

justify	Justify	XtJustify	XtJustifyCenter
label	Label	String	NULL
leftBitmap	LeftBitmap	Pixmap	None
mappedWhenManaged	MappedWhenManaged	Boolean	True
pointerColor	Foreground	Pixel	XtDefaultForeground
pointerColorBackground	Background	Pixel	XtDefaultBackground
resize	Resize	Boolean	True
sensitive	Sensitive	Boolean	True
width	Width	Dimension	text width
x	Position	Position	0
y	Position	Position	0

这里有一个在程序中设置几种资源的例子:

```
Widget label;
Arg args[5];
String app_resources[] =
    {"*Label.Label: Testing Athena Label Widget", NULL };
XtSetValues(args[0], XtNlabel, "The label of a widget");
XtSetValues(args[1], XtNwidth, 100);
XtSetValues(args[2], XtNheight, 90);
top_level = XtAppInitialize(&application_context, "Test", NULL, 0,
    &argc, argv,
    app_resources,
    NULL, 0);
label = XtCreateManagedWidget("label5", labelWidgetClass,
    top_level, args, 3);
```

你也可以在 Xdefaults 文件中设置资源属性值。例如, 假设程序的名称是 Test, 你想要设置标签部件的 x,y 坐标位置:

```
Test*label5*x:          5
Test*label5*y:          15
```

同样, 你也可以通过使用 X 工具包函数 XtGetValues 来获得一个部件的资源属性值。

```
Dimension width, height;
Arg args[2];
Widget label=XtCreateManagedWidget("label",labelWidgetClass,
    top_level,NULL,0);
XtSetArg(args[0],XtNwidth, &width);
XtSetArg(args[1],XtNheight, &height);
XtGetValues(label, args, 2);
printf("Widget width=%d and height=%d\n", width, height);
```

这里, 对变量 width 和 height 使用 Dimension 类型, 因为资源 width 和 height 具有类型 Dimension。同样, 应该使用 Pixel 类型来获得一个窗口部件的背景色。

25.4 XFree86

Linux 上最常用的 X 服务器是 XFree，它由 XFree86 Project Inc 开发。这种 X 服务器包含了一个 X 服务器所应该具有的所有基本功能。它还有一些额外的功能，它构建得很好足以和其他 X 服务器相媲美，并且是一种现代的图形环境。在本节中我们将简要介绍这些特性。

25.4.1 DPMS——显示器电源管理信令

DPMS 也称为 VESA 显示器电源管理信令标准 (Display Power Management Signaling Standard)，它用来启用电源管理支持。引用 VESA 的标准文件在这一问题上的所述 (显示器电源管理信令 (DPMS) 标准 1.0 版)：“提供在显示控制器和显示器之间的通信，并且对常用定义和方法进行标准化，这些方法让显示控制器向显示器发送信号启用显示器的多种电源管理状态。”

运行在 XFree 之下的图形驱动程序能够告诉 X Windows 它支持 DPMS 标准，并且规定可以获得哪种 DPMS 支持。DPMS 函数通常通过一个桌面 API 或清屏 API 来使用，而不是直接通过 X Windows 的 API 来使用。

25.4.2 DRI——直接显示接口

在 XFree86 的 4.0 版中引入了一种用于硬件加速三维图形的新接口。在编写本书的时候，还没有太多驱动程序完全支持它。但是，它对 Linux 上三维图形的未来有很大影响——不是说 Linux 上游戏的未来。你可以通过 OpenGL 调用来访问 DRI (Direct Render Interface) 加速的三维图形。如果你的程序使用 OpenGL，那么你要做的就是为图形卡安装一个 DRI 驱动程序，然后就能获得加速的三维图形效果。

要了解使用 OpenGL 的更多信息，请参考第 29 章“使用 OpenGL 和 Mesa 进行 3D 图形编程”。

25.4.3 DGA——直接图形体系结构

XFree 还支持直接访问 X 服务器的帧缓冲。通过 DGA (Direct Graphics Architecture) API，或者说直接图形体系结构得到这种支持。它对游戏和类似的需要以尽可能小的开销快速更新显示的程序最为有用。

不幸的是，并不是所有的 XFree 驱动程序都能使用 DGA 模式，在某些情况下这种支持有点古怪。因此，必须要在你的程序中支持回滚模式。一个工作速度慢但能工作的程序比完全不能工作的程序更好。

如果你正准备在 Linux 下编写游戏，可能最好从使用 Linux 上许多种多媒体库之一开始。即使底层的图形环境不同，这些库提供的 API 也保持了一致性。这意味着你可以在 X Windows、SVGA 库等系统上运行你的程序。在某些情况下，多媒体库甚至可以跨平台开发，从而支持其他操作系统，比如 Windows、Macintosh 和 BeOS。

要了解有关这些库的更多信息，可以参考：

http://www.libsdl.org/	简单的直接媒体层（Simple DirectMedia Layer）——一种跨平台的多媒体库，用于提供对图形帧缓冲设备和音频设备的快速访问。
http://www.ggi-project.org/	通用图形接口（General Graphics Interface）——一个开发项目，目标为开发一种可靠的、稳定的和快速的，能够随处工作的图形系统。

25.4.4 XV——X 视频

XFree 的 X 视频扩展让客户把视频当作任何其他原语来处理，因此能够让视频在图形资源（drawable）中显示。在编写本书的时候，这还是对 XFree 的一种新扩展，因此只支持把视频放入窗口类型的图形资源中。在未来发布的版本中，关于对 XV 扩展的讨论将能够用在其他种类的图形资源上。

25.5 小 结

在本章中，你学习了使用 Xlib 和 X 工具包进行 X Windows 编程的基础知识。你还可以在因特网上获得许多 X 窗口编程例子的源程序。在接下来的 3 章中，你会学到更高层的工具包，它们使编写 X Windows 程序的工作更加容易。但是，即使在你使用像 Athena、Motif、Qt 和 GTK+ 等高层工具包时，彻底理解 Xlib 的编程技术也是非常有价值的。

第 26 章 Athena、Motif 和 LessTif 窗口部件

你在第 25 章“X Windows 编程”中已经看到，使用低层的 Xlib API 编写简单的程序需要多么大的代码量。在那一章中，你也学到了 X 工具包，它是 Athena、Motif 以及大多数其他高层应用的窗口部件库所使用的一种工具库。

在本章里，你将开发一个短小的程序，这个程序显示了怎样使用标签、命令按钮、菜单和文本编辑框。

26.1 使用 Athena 的窗口部件

Athena 窗口部件集是为 MIT 的 Athena 计划编写的，该计划为 MIT 的学生和教职员提供了一种分布式的计算环境。许多程序员都使用 Athena 而不是 Motif，因为 Motif 不能免费得到。但是，最近这种情况有所改变，因为 Open Group 已经发布了 Motif 的源代码供非商业的开发人员免费使用。如果你只为 Linux 平台进行编程，那么可能会对使用较新的窗口部件库，比如 GTK+ 和 Qt 更感兴趣。但是，如果你想让自己的程序能够在尽可能多的 UNIX 平台上运行，那么仍然有理由使用 Athena 或 Motif。

Athena 窗口部件最初是平面的显示风格，但是绝大多数 Linux 发布版本都带有一种名为 awt3d 的可选软件包，它能够覆盖用于 Athena 窗口部件的库，把平面的显示风格替换成更吸引人的 3D 显示风格。

26.1.1 Athena 的标签窗口部件

本节的示例程序 label.c 需要下面的包含文件：

```
#include <X11/Intrinsic.h>
#include <X11/Xaw/Label.h>
```

Intrinsic.h 文件包含了 X 工具包的定义，而 Label.h 文件定义了 Athena 的 Label 类。一般的，X 应用程序使用你根目录下 Xdefaults 文件中的资源定义，这些资源定义是使你可以改变程序外观和行为的程序选项。这里，我们使用一个字符串数组来为本例中使用的 Label 窗口部件设置资源。Label 窗口部件具有一个 Label 属性，它的定义如下所示：

```
String app_resources[] =
    {"*Label.Label: Testing Athena Label Widget", NULL };
```

在主函数 main 中，我们必须定义三个变量：一个 Top Level 窗口部件，一个 Label 窗口部件和一个应用程序上下文：

```
XtAppContext application_context;
Widget top_level, label;
```

首先, 为应用程序创建一个 Top-Level 窗口部件并保存其值, 为变量 `application_context` 定义值:

```
top_level = XtAppInitialize(&application_context, "test", NULL, 0,
                           &argc, argv,
                           app_resources,
                           NULL, 0);
```

现在创建 Top-Level 窗口部件的一个子部件。这就是测试 Label 窗口部件:

```
label = XtCreateManagedWidget("label", labelWidgetClass,
                               top_level, NULL, 0);
```

在调用 `XtRealizeWidget` 之后, Top-Level 窗口部件和 Label 窗口部件将被处理为可见的。

```
XtRealizeWidget(top_level);    // create windows and make visible
```

最后, 将应用程序的上下文传递给 `XtAppMainLoop` 以处理这个简单程序的所有事件:

```
XtAppMainLoop(application_context);    // main event loop
```

正如在第 25 章中看到的, 程序 `label.c` 也包含了获取 Label 窗口部件 `width` 和 `height` 资源的例子代码:

```
Dimension width, height;
Arg args[2];
XtSetArg(args[0], XtNwidth, &width);
XtSetArg(args[1], XtNheight, &height);
XtGetValues(label, args, 2);
printf("Widget width=%d and height=%d\n", width, height);
```

图 26.1 显示了一个包含 `label.c` 示例程序的 X Window。

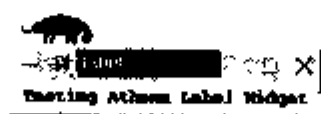


图 26.1 在 KDE 桌面环境上运行 `label.c` 示例程序

可以进入 `src/X/Athena` 目录下, 并键入:

```
make
./label
```

来建立并运行这个测试程序。

26.1.2 Athena 的命令按钮窗口部件

在这一节中我们要使用的示例程序 `button.c` 同上一节的 `label.c` 程序非常相似。它们源代码的不同之处在于 `button.c` 能让你设置命令按钮并管理事件的处理。

在 `button.c` 程序中, 需要包含下面的 `include` 文件来获得具有 `callback` 类型值的 `XtNcallback` 的常量定义:


```
NULL, 0);
```

与上一个例子 `label.c` 不同，这里想保存测试窗口部件的值（在这里，也就是 `Command Button` 窗口部件）：

```
command_button = XtCreateManagedWidget("command",
                                         commandWidgetClass, top_level, NULL, 0);
```

我们定义了一个回调函数 `do_button_clicked`。现在将这个回调函数分配给 `Command Button` 窗口部件：

```
XtAddCallback(command_button, XtNcallback, do_button_clicked,
              NULL);
```

最后，我们想让所有的窗口部件变为可见的并可以进行事件处理：

```
XtRealizeWidget(top_level);
XtAppMainLoop(application_context);
```

清楚地理解事件是如何进行处理的这一点非常重要。`Top-Level` 窗口部件包含了 `Command Button` 窗口部件。当 `Top-Level` 窗口部件变为可见时，X 服务器知道将该程序的事件同 `Top-Level` 窗口部件联系起来。当在命令行按钮上进行单击时，`XtAppMainLoop` 中的事件循环代码将从 X 服务器那里通过一个套接口连接获得有一个事件发生了的通知。`Top-Level` 窗口部件将这个事件向下传递给命令按钮，然后执行回调函数。

因为 X Windows 事件处理使用到了套接口，所以可以用 `netstat` 工具监视 X Window 此刻正在与什么套接口（也就是什么事件）一起工作。作为一个小实验，连续在终端窗口上运行 2 次 `netstat` 命令，但在你第一次运行 `netstat` 命令之后启动示例程序 `button.c`：

```
netstat -a > temp1
button &
netstat -a > temp2
diff temp1 temp2
rm temp1 temp2
```

上面的命令将显示出用来处理该应用程序的套接口连接。套接口的输入/输出在 Xlib 和 X 服务器（可能是远程的）之间进行。图 26.2 显示了一个包含示例程序 `button.c` 的 X 窗口。



图 26.2 在 KDE 桌面环境下运行示例程序 `button.c`

你可以通过进入 `src/X/Athena` 目录并键入下面的命令来建立并运行该测试程序：

```
make
./button
```

26.1.3 Athena 的列表窗口部件

List (列表) 窗口部件用来显示字符串列表。这些字符串在一个下拉框中显示, 每个字符串都可以用鼠标单击选中。这些窗口部件是许多更复杂的窗口部件, 比如文件选择窗口的基本组成单位。

本节示例程序 `list.c` 同前面两个示例程序 `label.c` 和 `button.c` 很相似。这里将集中处理列表窗口部件, 并假设你已经学习了前面的两节内容。下面的 `include` 文件定义了列表窗口部件:

```
#include <X11/Xaw/List.h>
```

使用 `button.c` 中的回调函数来处理鼠标单击事件。列表窗口部件的回调函数比较起来更加复杂一些, 因为我们想让列表窗口部件具有识别鼠标单击了哪一个列表项的功能。回调函数的第三个参数是一个依赖于窗口部件的数据。对于列表窗口部件来说, 第三个参数是一个 `XawListReturnStruct` 对象的地址, 这里我们对该对象中的两个域感兴趣, 它们如表 26.1 所示。

表 26.1 `XawListReturnStruct` 对象中的域

域名	描述
<code>int list_index</code>	被单击列表项的下标 (从 0 开始)
<code>char *string</code>	列表项的标签

回调函数 `do_list_item_selected` 用第三个参数来打印被单击列表项的下标和标签:

```
void do_list_item_selected(Widget w, XtPointer unused, XtPointer
    data) {
    XawListReturnStruct *list_item = (XawListReturnStruct*)data;
    printf("Selected item (%d) text is '%s'\n",
        list_item->list_index, list_item->string );
}
```

主函数 `main` 定义了两个窗口部件变量和一个应用程序上下文变量:

```
Widget top_level, list;
XtAppContext application_context;
```

创建一个 Top-Level 窗口部件, 并为应用程序上下文填充数据:

```
top_level = XtAppInitialize(&application_context, "listexample",
    NULL, ZERO,
    &argc, argv, NULL,
    NULL, 0);
```

当创建列表窗口部件时, 我们为列表项的标签提供 “`char *`” 类型的字符串数组:

```
String items[] = {
    "1", "2", "3", "4", "5", "six", "seven", "8",
    "9'th list entry", "this is the tenth list entry",
```

```

    "11", "12",
    NULL
};

```

为了创建一个列表窗口部件，使用 `XtVaCreateManagedWidget`——`XtCreateManagedWidget` 的另一种形式，它接受可变数量的选项：

```

list= XtVaCreateManagedWidget("list", listWidgetClass, top_level,
                                XtNlist, items,
                                NULL, 0);

```

我们需要将该回调函数与列表窗口部件按 `button.c` 例子中的方式联系起来：

```

XtAddCallback(list, XtNcallback,
              do_list_item_selected, (XtPointer)NULL);

```

和前面一样，将窗口部件在 X 服务器上变为可见的并可以进行事件处理：

```

XtRealizeWidget(top_level);
XtAppMainLoop(application_context);

```

图 26.3 显示了一个包含 `list.c` 示例程序的 X 窗口。

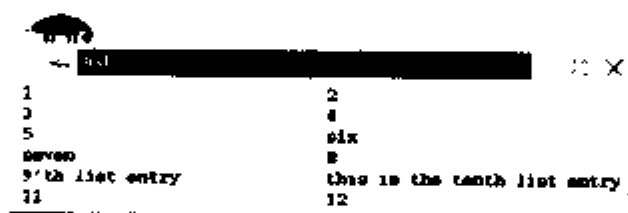


图 26.3 在 KDE 桌面环境下运行 `list.c` 示例程序

你可以通过进入 `src/X/Athena` 目录并键入下面的命令来建立并运行该测试程序：

```

make
./list

```

26.1.4 Athena 的文本窗口部件

文本窗口部件提供了一个输入框，用户可以在其中输入简短的文本。Athena 的文本窗口部件具有许多可通过使用资源属性值（或者在 `Xdefaults` 文件中，或者使用程序定义的默认值）进行设置的选项。下面的 3 个 `include` 文件分别定义了窗格（`Paned`），ASCII 文本（`AsciiText`）和命令按钮（`Command button`）窗口部件。`Paned` 是一个用来包含其他窗口部件的容器部件。默认情况下，`Paned` 提供可以用来手动调整任何被包含窗口部件大小的“小手”工具。

```

#include <X11/Xaw/Paned.h>
#include <X11/Xaw/AsciiText.h>
#include <X11/Xaw/Command.h>

```

当定义一个文本窗口部件时，我们想通过回调函数在窗口部件中显示文本以及清除窗口部件中的所有文本。为了显示窗口部件中的文本，将使用 X 工具包中的 `XtVaGetValues`

函数，它将对一个窗口部件（被指定为该函数的第一个参数）进行操作。第二个参数将是 `XtNstring` 常量，它用来说明我们想要检索字符串资源的属性值。第三个参数是一个字符串变量的地址值，它将被设置为指向一块字符数据。该字符数据块的存储由文本窗口部件内部进行管理。第四个参数是 `NULL`，用来指明没有其他可从文本窗口部件中检索的资源属性值了。

下面的代码列表显示了 `do_display_widget_text` 回调函数的详细实现。该函数在文本窗口部件中发生事件时被调用。为了演示的目的，使用 `XtVaGetValues` 来获得文本窗口部件中的当前文本，然后打印该文本：

```
void do_display_widget_text(Widget w,
                           XtPointer text_ptr,
                           XtPointer unused) {
    Widget text = (Widget) text_ptr;
    String str;
    XtVaGetValues(text,
                  XtNstring, &str,
                  NULL);
    printf("Widget Text is:\n%s\n", str);
}
```

我们还想清除文本窗口部件中的所有文本。为此，使用 X 工具包的 `XtVaSetValues` 函数，将字符串资源的值设置为 `NULL`：

```
void do_erase_text_widget(Widget w,
                          XtPointer text_ptr,
                          XtPointer unused) {
    Widget text = (Widget) text_ptr;
    XtVaSetValues(text,
                  XtNstring, "",
                  NULL);
}
```

这里再向示例程序中加入一些新的东西：一个退出按钮。可以简单地直接调用 `exit`，但在退出之前最好首先调用 `XtDestroyApplicationContext` 以清除任何对共享 X 资源的访问。在这个例子中，将应用程序上下文定义为一个全局变量，这样创建退出按钮的 `do_quit` 回调函数便可以访问应用程序上下文了：

```
XtAppContext application_context;
void do_quit(Widget w, XtPointer unused1, XtPointer unused2) {
    XtDestroyApplicationContext(application_context);
    exit(0);
}
```

和前面例子一样，直接在程序中定义默认的应用程序资源。在下面的代码中，为 `Text`（文本）类、`erase`（清除）类和 `display`（显示）类定义资源。注意到我们没有像对 `erase` 和 `display` 命令按钮一样，为 `Quit`（退出）命令按钮定义任何默认资源。`Quit` 按钮将以它的名字作为其默认标签——在这个例子中，就是退出（Quit）。

下面的代码显示了如何设置默认的应用程序资源:

```
String app_resources[] = {
    "**Text*editType: edit",
    "**Text*autoFill: on",
    "**Text*scrollVertical: whenNeeded",
    "**Text*scrollHorizontal:whenNeeded",
    "**erase*label: Erase the Text widget",
    "**display*label: Display the text from the Text widget",
    "**Text*preferredPaneSize: 300",
    NULL
};
```

文本窗口部件资源属性 `autoFill` 用来控制自动换行。文本窗口部件属性 `editType` 用来设置文本为可编辑的。属性 `preferredPaneSize` 用来为包含在窗格窗口部件中的其他窗口部件设置以像素计算的目标高度值。主函数 `main` 定义了 Top-Level 窗口部件以及文本区的窗口部件, 包括清除命令按钮、显示命令按钮和退出命令按钮:

```
Widget top_level, paned, text, erase, display, quit;
```

初始化文本窗口部件, 使用下面的初始文本:

```
char *initial_text= "Try typing\n\nsome text here!\n\n";
```

将应用程序上下文进行初始化 (这里将应用程序上下文定义为一个全局变量, 从而可以在 `do_quit` 函数中对它进行访问), 用和前面的例子中一样的方法定义 Top-Level 窗口部件, 并创建一个面板窗口部件:

```
top_level = XtAppInitialize(&application_context,
                           "textexample", NULL, 0,
                           &argc, argv, app_resources,
                           NULL, 0);
paned = XtVaCreateManagedWidget("paned", panedWidgetClass,
                                top_level, NULL);
```

使用函数 `XtCreateManagedWidget` 的可变参数版本来创建文本窗口部件, 这样既可以指定窗口部件的类型 `XawAsciiString`, 还可以通过设置 `type` 和 `string` 属性来设置初始文本:

```
text = XtVaCreateManagedWidget("text", asciiTextWidgetClass, paned,
                                XtNtype, XawAsciiString,
                                XtNstring, initial_text,
                                NULL);
```

这里, 使用 `paned` 作为父窗口部件, 而不是 `Top_Level` 窗口部件。同样, 为了创建标记为“erase”的命令窗口部件, 可以使用普通版本的 `XtCreateManagedWidget` 函数, 因为我们并不设置任何属性。然而, 这里调用了一个支持可变参数的函数版本 (将 `paned` 作为父窗口部件):

```
erase = XtVaCreateManagedWidget("erase", commandWidgetClass, paned,
                                NULL);
```

以同样的方式，定义 `display` 和 `erase` 窗口部件，然后为所有的 3 个 `Command Button` 窗口部件设置回调函数。使用 `paned` 作为父窗口部件，而不是 `Top-Level` 窗口部件：

```
display = XtVaCreateManagedWidget("display", commandWidgetClass,
                                   paned, NULL);

quit = XtVaCreateManagedWidget("quit", commandWidgetClass, paned,
                                NULL);

XtAddCallback(erase, XtNcallback,
              do_erase_text_widget, (XtPointer) text);
XtAddCallback(display, XtNcallback, do_display_widget_text,
              (XtPointer) text);
XtAddCallback(quit, XtNcallback, do_quit, (XtPointer) text);
```

最后，和前面的例子一样，使所有的窗口部件在 X 服务器上变为可见的，然后进入主事件循环：

```
XtRealizeWidget(top_level);
XtAppMainLoop(application_context);
```

图 26.4 显示了一个包含 `text.c` 示例程序的 X 窗口。

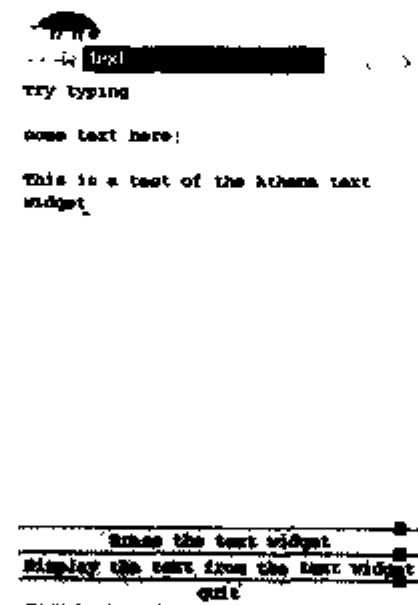


图 26.4 在 KDE 桌面环境下运行 `text.c` 示例程序

可以通过进入 `src/X/Athena` 目录并键入下面的命令来建立并运行该测试程序：

```
make
./text
```

26.1.5 Athena 的简单菜单窗口部件

本节示例程序是 `menu.c`，存放在 `src/X/Athena` 目录下。我们可以联合使用 3 个 Athena 窗口部件来创建一个简单的菜单，这 3 个窗口部件如表 26.2 所示。

表 26.2 3 个 Athena 窗口部件

窗口部件	描述
MenuButton	实现一个管理简单的弹出式菜单 shell 的菜单按钮窗口部件
SimpleMenu	弹出式 shell 和菜单元素的容器
Sme	被加到一个简单菜单的简单菜单元素

下面的 include 文件定义了上述窗口部件类型：

```
#include <X11/Xaw/MenuButton.h>
#include <X11/Xaw/SimpleMenu.h>
#include <X11/Xaw/Sme.h>
#include <X11/Xaw/SmeBSB.h>
```

被加到一个简单菜单的每一个菜单元素可以拥有自己惟一的、用以执行特定菜单动作的回调函数。在 menu.c 程序中，我们使用一个简单的回调函数 do_menu_selection，它使用菜单元素的名字来决定用户具体选择了哪一个菜单元素。示例程序 menu.c 有一个标记为“Exit program”的菜单元素。因为想在退出程序之前清除任何 X 资源和所使用的资源，我们将应用程序上下文定义为一个全局变量，从而它可以在回调函数中使用：

```
XtAppContext application_context;

void do_menu_selection(Widget item_selected, XtPointer unused1,
                      XtPointer unused2) {
    char * name = XtName(item_selected);
    printf("Menu item '%s' has been selected.\n", name);
    if (strcmp(name, "Exit program") == 0) {
        XtDestroyApplicationContext(application_context);
        exit(0);
    }
}
```

XtName 宏从一个窗口部件中返回它的名字域。该名字可以用来确定在运行示例程序时，用户选择了哪一个菜单元素。虽然我们通常使用 Xdefaults 文件来定制 X 应用程序资源，出现在 menu.c 例子中的下面的数据可用来定义默认的资源：

```
String fallback_resources[] = {
    "*menuButton.label: This is a menubutton label",
    "*menu.label: Sample menu label",
    NULL,
};
```

该数据在示例程序中被作为全局变量进行分配，它也可以在主函数中进行定义。我们在主函数 main 中定义菜单项标签：

```
char * menu_item_names[] = {
    "Menu item1", "Menu item2", "Menu item3",
    "Menu item4", "Exit program",
};
```

在主函数中分配窗口部件的数据是合适的。然而，你必须小心：如果你使用一个分离的、由主函数调用的“设置”函数，请确保要将任何窗口部件设置数据都声明为静态的，这样在程序退出该函数时，数据的内存分配才不会丢失。（记住：函数中的非静态数据被分配在调用堆栈上，该存储在退出函数时将丢失。）我们已经定义了4个用来存储在 menu.c 例程中创建的窗口部件值的变量：

```
Widget top_level, menu_button, menu, menu_element;
```

变量 menu_element 将为每一个创建并加入到一个简单菜单的菜单元素所重用（例如，一个 Sme 窗口部件）。正如在前面的例子中所看到的，我们需要创建应用程序的窗口部件：

```
top_level = XtVaAppInitialize(&application_context,
                             "textmenu", NULL, 0,
                             &argc, argv, fallback_resources,
                             NULL);

menu_button = XtCreateManagedWidget("menuButton",
                                     menuButtonWidgetClass,
                                     top_level, NULL, 0);

menu = XtCreatePopupShell("menu", simpleMenuWidgetClass,
                          menu_button, NULL, 0);
```

现在我们在每一个菜单元素的名字间循环，创建一个新的 Sme 窗口部件并将其加入到该简单菜单。同样将回调函数分配给每一个菜单元素：

```
for (i = 0; i < (int)XtNumber(menu_item_names) ; i++) {
    menu_element = XtCreateManagedWidget(menu_item_names[i],
                                           smeBSBObjectClass,
                                           menu, NULL, 0);
    XtAddCallback(menu_element, XtNcallback, do_menu_selection,
                  NULL);
}
```

这里，宏 XtNumber 提供了在 menu_item_names 数组中定义的非空字符串的个数。每一个菜单元素的父窗口部件被设置为变量 menu（也就是我们的简单菜单窗口部件）。你可以进入 src/X/Athena 目录并键入下面的命令来建立并运行该测试程序：

```
make
./menu
```

26.2 使用 Motif 的窗口部件

Linux 上有几种版本的 Motif 窗口部件。Open Group 最近发布了源代码供在非商业的程序中免费使用 Motif。Motif 窗口部件库有几种商业实现。还有一种真正免费的 Motif 克隆体，它叫作 LessTif（参见 <http://www.lesstif.org>）。这个软件按照 LGPL 许可证发布，因此也可以用于商业编程。

本节中的程序都使用 LessTif 开发并经过测试。但是，在任何其他的 Motif 版本中，它们应该不会产生任何问题。

在这一节，将使用 3 个示例程序（它们都位于 src/X/Motif 目录下）展示 Motif 窗口部件的创建过程。这些程序如表 26.3 所示。

表 26.3 展示 Motif 窗口部件创建过程的 3 个示例程序

程序	描述
label.c	介绍 Motif
list.c	显示怎样处理事件
text.c	将带有文本和命令按钮窗口部件和面板窗器结合起来

我们所学的大部分关于 Athena 窗口部件的使用将同样适用于 Motif 程序。使用 Motif 的最大不同可能在于它使用 Motif 字符串代替了每个字符 8 比特的 C 字符串。另外一个不同在于窗口部件资源的命名和定义。Athena 窗口部件的头文件为在头文件中定义的部件定义新的资源类型。Motif 窗口部件资源的名字在 include 文件 XmStrDefs.h 中定义。下面是一些 Motif 资源名字的实例：

```
#define XmNalignment "alignment"
#define XmNallowOverlap "allowOverlap"
#define XmNallowResize "allowResize"
#define XmNclientData "clientData"
#define XmNclipWindow "clipWindow"
#define XmNcolumns "columns"
#define XmNcommand "command"
#define XmNdirListItemCount "dirListItemCount"
#define XmNdirListItems "dirListItems"
#define XmNeditable "editable"
#define XmNlabelString "labelString"
#define XmNoffsetX "offsetX"
#define XmNoffsetY "offsetY"
#define XmNwidth XtNwidth // define using X toolkit constant
#define XmNheight XtNheight // define using X toolkit constant
```

这些只是可用的 Motif 窗口部件资源的一些例子。本节 Motif 编程介绍非常短，因此这里我们将仅涉及少数几个 Motif 窗口部件。

26.2.1 Motif 的标签窗口部件

本节示例程序是 src/X/Motif 目录下的 label.c 文件。该例子程序使用两个 include 文件，一个用来为所有的 Motif 窗口部件做核心定义，另外一个用来定义 Motif 的标签窗口部件：

```
#include <Xm/Xm.h>
#include <Xm/Label.h>
```

主函数 main 定义了两个窗口部件，一个是 Top-Level 应用程序窗口部件，一个是标签窗口部件。注意这里使用了和上面的 Athena 窗口部件示例程序中同样的数据类型。下面的代码显示了一个 Motif 字符串的定义，以及一个用来设置该字符串标签资源的参数变量：

```
Widget top_level, label;
XmString motif_string;
Arg arg[1];
```

这里，定义了一个 Top-Level 应用程序窗口部件。该例子与 Athena 窗口部件例子程序不同，因为我们没有定义应用程序上下文。其实可以按在 Athena 窗口部件的所有例子中完全一样的方式来编写 Top-Level 窗口部件的初始化代码。

```
top_level = XtInitialize(argv[0], "test", NULL, 0, &argc, argv);
```

我们不能简单地给 Motif 字符串资源使用 C 字符串。实际上，必须构造一个 Motif 字符串：

```
motif_string = XmStringCreateSimple("Yes! we are testing the Motif
                                   Label Widget!");
```

这里，为标签窗口部件设置 labelString 资源，然后构造一个标签窗口部件：

```
XtSetArg(arg[0], XmNlabelString, motif_string);
label = XmCreateLabel(top_level, "label", arg, 1);
XtManageChild(label);
```

和在 Athena 窗口部件例子中一样，需要使 Top-Level 窗口部件变为可见的、可以处理的事件：

```
XtRealizeWidget(top_level);
XtMainLoop();
```

这里，使用 X 工具包函数 XtMainLoop 来处理事件，因为在创建 Top-Level 应用程序窗口部件时，没有定义应用程序上下文。Athena 窗口部件程序中使用 XtAppMainLoop (XtAppContext application_context) 函数来处理 X 事件。如果需要一个应用程序的应用程序上下文，可以使用下面的函数，将该应用程序中的任何窗口部件作为参数即可：

```
XtAppContext XtWidgetToApplicationContext(Widget w)
```

图 26.5 显示了 label.c 例子程序在 KDE 桌面环境下运行的情况。

可以通过进入 src/X/Motif 目录并键入下面的命令来建立并运行该测试程序：

```
make
./label
```



图 26.5 Motif 标签窗口部件实例

26.2.2 Motif 的列表窗口部件

本节示例程序是 src/X/Motif 目录下的 list.c 文件。该程序显示了如何创建一个 Motif 的列表窗口部件以及如何回调处理。它使用两个 include 文件，一个用来为所有的 Motif 窗口部件做核心定义，另外一个用来定义 Motif 的列表窗口部件：

```
#include <Xm/Xm.h>
#include <Xm/List.h>
```

我们需要一个回调函数来处理列表窗口部件中的用户选择事件：

```
void do_list_click(Widget widget, caddr_t data1, XtPointer data2){
    char *string;
    XmListCallbackStruct *callback = (XmListCallbackStruct *)data2;
    XmStringGetLtoR(callback->item, XmSTRING_OS_CHARSET, &string);
    printf(" You chose item %d : %s\n", callback->item_position,
           string);
    XtFree(string);
}
```

当 `do_list_click` 函数被 `XtMainLoop` 中的事件处理程序调用时，传递的第三个参数是一个指向 `XmListCallbackStruct` 对象的指针。在这个例子中，我们将第三个参数强制转换为 `XmListCallbackStruct *` 类型，并使用具有 `XmString` 类型的数据项域。Motif 实用函数 `XmStringGetLtoR` 用来将一个 Motif 字符串转换成一个使用默认字符集的 C 字符串。函数 `XmStringGetLtoR` 为该 C 类型的字符串分配存储空间，因此在不使用该字符串时，需要使用 `XtFree` 函数来释放该存储空间。

主函数 `main` 定义了两个窗口部件、一个包含三个 Motif 字符串的数组以及一个包含四个参数变量的数组：

```
Widget top_level, list;
XmString motif_strings[3];
Arg arg[4];
```

我们定义了一个 Top-Level 应用程序窗口部件，不带额外参数：

```
top_level = XtInitialize(argv[0], "test", NULL, 0, &argc, argv);
```

接着，为 Motif 字符串数组中的每一个元素定义了取值，并在创建 Motif 的列表窗口部件时使用了该数组：

```
motif_strings[0] = XmStringCreateSimple("list item at index 0");
motif_strings[1] = XmStringCreateSimple("list item at index 1");
motif_strings[2] = XmStringCreateSimple("list item at index 2");

XtSetArg(arg[0], XmNitemCount, 3);
XtSetArg(arg[1], XmNitems, motif_strings);
XtSetArg(arg[2], XmNvisibleItemCount, 3);
// all list elements are visible
XtSetArg(arg[3], XmNselectionPolicy, XmSINGLE_SELECT);

list = XmCreateList(top_level, "list", arg, 4);
```

我们已经指定了 4 个参数用来创建列表窗口部件：

- 列表元素的个数
- 列表元素的数据

- 应该是可见的列表元素的个数
- 一次只有一个列表元素可以选择

如果允许多个列表元素选择，那么就应该重新编写回调函数 `do_list_click` 来处理对多个列表元素的检索。在创建列表窗口部件之后，我们为该窗口部件指定回调函数 `do_list_click`，使列表窗口部件和 Top-Level 窗口部件成为可见的，并可以处理 X 事件：

```
XtAddCallback(list, XmNsingleSelectionCallback,
              (XtCallbackProc)do_list_click, NULL);
XtManageChild(list);
XtRealizeWidget(top_level);
XtMainLoop();
```

图 26.6 显示了 `list.c` 例子程序在 KDE 桌面环境下运行的情况。



图 26.6 Motif 列表窗口部件示例

你可以通过进入 `src/X/Motif` 目录并键入下面的命令来建立并运行该测试程序：

```
make
./list
```

使用 Motif 窗口部件比使用 Athena 窗口部件稍微复杂一些，但总体来看，使用 Motif 窗口部件拥有更多的可用选项。在过去的 10 年中，我使用了 Athena 和 Motif 窗口部件，并基于应用的需求和客户的倾向来选择具体使用哪种窗口部件集。

26.2.3 Motif 的文本窗口部件

本节示例程序是 `src/X/Motif` 目录下的 `text.c` 文件。该程序介绍了用来包含其他窗口部件的面板容器的使用。程序中使用了一个文本窗口部件和两个按钮窗口部件。本程序仅需要两个 `include` 文件，一个用来为所有的 Motif 窗口部件做核心定义，另外一个用来定义 Motif 的文本窗口部件。我们使用一个 Motif 实用函数来创建 Motif 按钮窗口部件和面板窗口部件，因此我们不再需要单独的 `include` 文件来支持这些窗口部件类型：

```
#include <Xm/Xm.h>
#include <Xm/Text.h>
```

该程序使用了两个回调函数——分别与每个按钮窗口部件对应。第一个回调函数被用来显示文本窗口部件中的文本：

```
void do_get_text(Widget widget, caddr_t client_data, caddr_t
                 call_data){
    Widget text = (Widget)client_data;
    char *string = XmTextGetString(text);
```

```

    printf("The Motif example Text widget contains:\n%s\n", string);
    XtFree(string);
}

```

这是第一个使用第二个参数的回调函数例子。当将此回调函数与“display text”按钮窗口部件进行绑定时，我们指定文本窗口部件应该被作为客户端数据传递给回调函数。继续看示例程序代码：

```

Widget text = XmCreateText(pane, "text", NULL, 0);
Widget display_button = XmCreatePushButton(pane, "Display", NULL, 0);
XtAddCallback(display_button, XmNactivateCallback,
               (XtCallbackProc)do_get_text, text);

```

这里，文本窗口部件的属性值被指定为回调函数的客户端数据。

第二个回调函数要简单一些：它仅仅调用系统函数 `exit` 来终止整个程序：

```

void do_quit(Widget widget, caddr_t client_data, caddr_t call_data)
{
    exit(0);
}

```

主函数 `main` 定义了 5 个需要使用的窗口部件：

```

Widget top_level, pane, text, display_button, quit_button;

```

`pane` 窗口部件将被加入 Top-Level 应用程序窗口部件。所有其他的窗口部件将被加入到 `pane` 窗口部件中，`pane` 将文本和按钮窗口部件垂直排放：

```

top_level = XtInitialize(argv[0], "test", NULL, 0, &argc, argv);
pane = XmCreatePanedWindow(top_level, "pane", NULL, 0);

```

在这个例子中，在创建文本窗口部件之前为文本窗口部件的 5 个资源设置属性值：

```

XtSetArg(arg[0], XmNwidth, 400);
XtSetArg(arg[1], XmNheight, 400);
XtSetArg(arg[3], XmNwordWrap, TRUE);
XtSetArg(arg[4], XmNeditMode, XmMULTI_LINE_EDIT);
text = XmCreateText(pane, "text", arg, 5);

```

我们想指定文本编辑区的大小。面板窗口部件将自动调整其大小到它包含的窗口部件要求的最佳值。将文本窗口部件的宽度设置为 400 像素点将有效地设置面板窗口部件的宽度为 400 像素点，因为两个按钮窗口部件的最佳大小将小于 400 像素点宽，除非它们在创建时使用了非常长的标题。我们指定文本窗口部件使用字换行，从而对于长的行可以自动换到文本的下一行。

这里，将两个按钮定义为 `pane` 窗口部件的子窗口部件，并指定它们的回调函数：

```

display_button = XmCreatePushButton(pane, "Display", NULL, 0);
quit_button = XmCreatePushButton(pane, "Quit", NULL, 0);
XtAddCallback(display_button, XmNactivateCallback,

```

```
(XtCallbackProc)do_get_text, text);  
XtAddCallback(quit_button, XmNactivateCallback,  
              (XtCallbackProc)do_quit, NULL);
```

正如在前面所提醒过的，我们将文本窗口部件的值作为客户数据传递给“display text”按钮窗口部件的回调函数。在这个例子中，在使 Top-Level 窗口部件变为可见之前，我们单独管理每一个窗口部件。然后调用 XtMainLoop 进行事件处理：

```
XtManageChild(text);  
XtManageChild(display_button);  
XtManageChild(quit_button);  
XtManageChild(pane);  
XtRealizeWidget(top_level);  
XtMainLoop();
```

图 26.7 显示了 text.c 例子程序在 KDE 桌面环境下运行的情况。

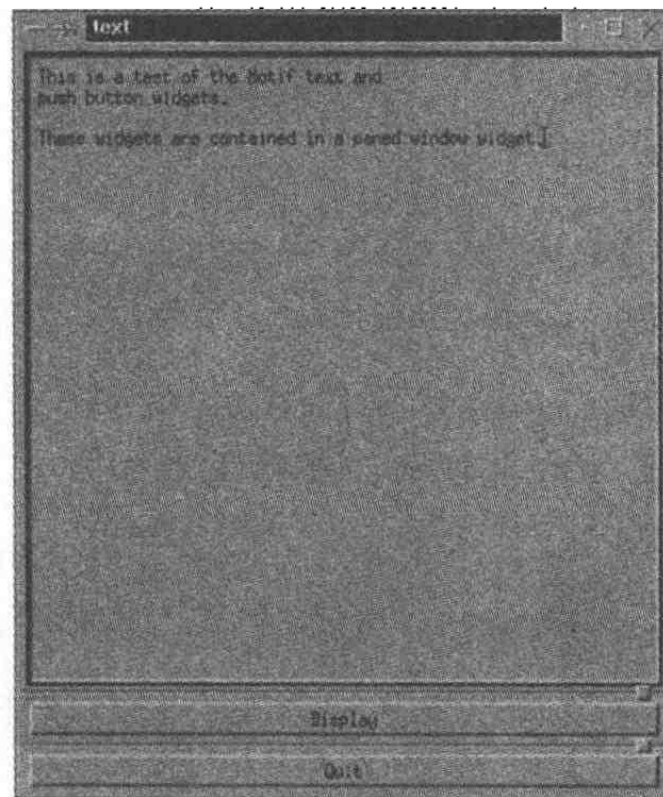


图 26.7 Motif 文本窗口部件示例

可以通过进入 src/X/Motif 目录并键入下面的命令来建立并运行该测试程序：

```
make  
./text
```

该例子显示了大多数 Motif 应用程序的标准结构：定义一个包含应用程序中所有其他窗口部件的面板窗口部件。在用户改变了主应用程序窗口的大小时，该面板窗口部件将处理子窗口部件的重画。另一个做法是将子窗口部件直接放到主应用程序窗口部件中，并指定每一个子窗口部件的 x-y 坐标和大小。

26.3 编写一个定制的 Athena 窗口部件

许多 X Windows 程序员从不需要编写定制的窗口部件，但定义新窗口部件的能力是一种封装应用程序数据和行为的非常强大的技术。在这一节中，将创建一个名为 URLWidget 的 Athena 窗口部件，用来从一个 URL 地址显示文本。编写新的窗口部件看上去好像是一个非常繁重的任务，但这一工作可以通过寻找一个类似窗口部件的源代码并对其进行修改而变得容易。

对于本例来说，将从 X 协会的 Template 例子部件开始。你也可以去他们的网站 <http://www.x.org> 看一看。

URLWidget 的源代码和一个测试程序 test.c 都存放在 src/X/URLWidget 目录下。可将用来作为编写自己的 Athena 窗口部件起点的原始 Template 例子存放在目录 src/X/URLWidget/templates 下。URLWidget 是一个简单的窗口部件，它从给定 URL 网址的文件中读取文本数据，然后将文本以 C 字符串的形式存放在该部件的私有数据区。每当该窗口部件需要重画时，对这个字符串进行分析，提取出其中独立的行，然后在部件的可绘制区域进行绘制。有 4 个文件用来实现该窗口部件，如表 26.4 所示。

表 26.4 实现 URLWidget 窗口部件的 4 个文件

文件	描述
fetch_url.c	一些用来连接 Web 服务器以及请求文件的套接口 (socket) 代码
URL.h	当要创建一个 URLWidget 时在程序中需要包含的公共头文件
URLP.h	私有的、用于具体实现的头文件
URL.c	URLWidget 的具体实现

文件 URL.h、URLP.h 和 URL.c 分别基于文件 Template.h、TemplateP.h 和 Template.c，后者带有下面的版权信息：

```
/* X Consortium: Template.c, v 1.2 88/10/25 17:40:25 swick Exp $ */
/* Copyright Massachusetts Institute of Technology 1987, 1988 */
```

一般情况下，只要所有的版权信息保留不变，X 协会允许免费使用他们的软件。

26.3.1 使用 fetch_url.c 文件

用于获取一个远端 Web 文档实用工具在 fetch_url.c 文件中的实现，该文件存放在 src/X/URLWidget 目录下，是从 src/IPC/SOCKETS 目录下的 web_client.c 文件所得。该示例程序在第 19 章中已经讨论过。这个实用函数的原型定义如下所示：

```
char * fetch_url(char *url_name);
```

下面是一些 fetch_url 用法的具体例子：

```
char * mark_home_page1 = fetch_url("www.markwatson.com");
char * mark_home_page2 = fetch_url("http://www.markwatson.com");
```

```
char * mark_home_page3 = fetch_url("http://www.markwatson.com
                                   /index.html");
```

对该实用工具函数接下来的讨论将非常简洁。如果需要可以参考第 19 章中的有关内容。Web 服务器在默认情况下倾听端口 80:

```
int port = 80;
```

表 26.5 列出了部分字符数组及其相应的用法。

表 26.5 字符数组及其相应的用法

字符数组	描述
<code>char* buf=(char*)malloc(50000)</code>	用于返回数据
<code>char message[256]</code>	用来建立一个发给 Web 服务器的“GET”消息
<code>char host_name[200]</code>	用来表示从指定的 URL 构造出的合适主机名
<code>char file_name[200]</code>	用来表示 URL 地址末端的可选文件名
<code>char *error_string</code>	用来构造一个合适的错误消息

该实用函数分配内存块并将其返回给调用者，但释放这些内存资源则是调用者自己的责任。下面的代码用来确定正确的主机名和 URL 地址末端可选的文件名:

```
char *sp1, *sp2;
sp1 = strstr(url, "http://");
if (sp1 == NULL) {
    sprintf(host_name, "%s", url);
} else {
    sprintf(host_name, "%s", &(url[7]));
}
printf("1. host_name=%s\n", host_name);
sp1 = strstr(host_name, "/");
if (sp1 == NULL) {
    // no file name, so use index.html:
    sprintf(file_name, "index.html");
} else {
    sprintf(file_name, "%s", (char *) (sp1 + 1));
    *sp1 = '\0';
}
printf("2. host_name=%s, file_name=%s\n", host_name, file_name);
```

现在建立了一个通向远端 Web 服务器的连接:

```
if ((nlp_host = gethostbyname(host_name)) == 0) {
    error_string = (char *) malloc(128);
    sprintf(error_string, "Error resolving local host\n");
    return error_string;
}
bzero(&pin, sizeof(pin));
```

```

pin.sin_family = AF_INET;
pin.sin_addr.s_addr = htonl(INADDR_ANY);
pin.sin_addr.s_addr = ((struct in_addr *) (nlp_host->h_addr))
    ->s_addr;
pin.sin_port = htons(port);
if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    error_string = (char *)malloc(128);
    sprintf(error_string, "Error opening socket\n");
    return error_string;
}
if (connect (sd, (void *)&pin, sizeof(pin)) == -1) {
    error_string = (char *)malloc(128);
    sprintf(error_string, "Error connecting to socket\n");
    return error_string;
}

```

然后，为服务器构造一个合适的 GET 消息并发送出去：

```

sprintf(message, "GET /%s HTTP/1.0\n\n", file_name);
if (send(sd, message, strlen(message), 0) == -1) {
    error_string = (char *)malloc(128);
    sprintf(error_string, "Error in send\n");
    return error_string;
}

```

现在等待响应，Web 服务器可能以几个分离的数据包返回数据：

```

count = sum = iter = 0;
while (iterr++ < 5) {
    count = recv(sd, &(buf[sum]), 50000 - count, 0);
    if (count == -1) {
        break;
    }
    sum += count;
}

```

最后，将套接口连接关闭（如果 Web 服务器还没有关闭该连接），然后从指定的 URL 返回数据：

```

close(sd);
return buf;

```

26.3.2 使用 URL.h 文件

URL.h 文件是 URLWidget 窗口部件的公共头文件。它是从 X 协会的 Template.h 文件变化得来的。该文件定义了 URLWidget 窗口部件的资源名：

```

#define XtNURLResource          "urlResource"
#define XtCURLResource          "URLResource"

```

例如，下面将可以看到（在测试程序 test.c 中）如何设置 URLWidget 的 URLResource 属性：

```
XtSetArg(args[2], XtNURLResource, "www.knowledgebooks.com");
```

下面的结构定义被用来实现 URLWidget 窗口部件：

```
typedef struct _URLRec* URLWidget;
```

下面的声明用来定义实际构造一个 URLWidget 窗口部件的部件类变量：

```
extern WidgetClass urlWidgetClass;
```

例如，在 test.c 程序中：

```
Widget url = XtCreateManagedWidget("url", urlWidgetClass, top_level,
                                     args, 3);
```

26.3.3 使用 URLP.h 文件

URLP.h 文件是作为 URLWidget 窗口部件的私有头文件。它是从 X 协会的 TemplateP.h 文件变化得来的。该私有头文件需要公共头文件和核心窗口部件定义头文件中的定义：

```
#include "URL.h"
#include <X11/CoreP.h>
```

我们需要定义一个惟一的、在 X11 的 StringDefs.h 文件中没有使用过的表示类型：

```
#define XtURLResource "URLResource"
```

每一个窗口部件都有核心数据和类数据。这里我们必须给出类数据结构的定义，即使该窗口部件并不需要任何类数据：

```
typedef struct {
    int empty;
} URLClassPart;
```

下面的代码定义了类记录（class record）数据，由核心（默认）数据和类数据组成：

```
typedef struct _URLClassRec {
    CoreClassPart core_class;
    URLClassPart URL_class;
} URLClassRec;
```

下面的外部符号引用将在 URL.c 文件中进行定义：

```
extern URLClassRec urlClassRec;
```

下面的结构定义了该窗口部件类型特有的资源：

```
typedef struct {
    /* resources */
    char* name;
    /* private state */
}
```

```

    char *data;
    GC gc;
} URLPart;

```

URLPart 结构的私有状态数据将在 URL.c 文件的 Initialize 函数中进行初始化, 并在 ReDraw 函数中使用。每当设置 XtNURLResource 资源时, 资源数据的名字也将被设置。下面的结构包含了 URLPart 数据:

```

typedef struct _URLRec {
    CorePart    core;
    URLPart     url;
} URLRec;

```

在下面将看到, Initialize 和 Redraw 函数接受一个 URLWidget 对象作为参数, 而且它们通过窗口部件指针来访问 URLPart 数据:

```

static void Initialize(URLWidget request, URLWidget new) {
    new->url.data = fetch_url(new->url.name);
}
static void ReDraw(URLWidget w, XEvent *event, Region region) {
    XDrawString(XtDisplay(w), XtWindow(w), w->url.gc, 10, y,
                "test", strlen("test"));
}

```

26.3.4 使用 URL.c 文件

URL.c 文件包含了 URLWidget 窗口部件的具体实现, 是从 X 协会的 Template.c 文件变化得来的。该实现需要一些头文件, 其中最主要的是 URLWidget 类的私有头文件:

```

#include <X11/IntrinsicP.h>
#include <X11/StringDefs.h>
#include "URLP.h"

```

依据 Template.c 程序, 我们需要进行资源定义, 这样程序才可以使用 XtNURLResource 类:

```

static XtResource resources[] = {
#define offset(field) XtOffset(URLWidget, url.field)
    /* {name,class,type,size,offset,default_type,default_addr},*/
    { XtNURLResource, XtCURLResource, XtRURLResource, sizeof(char *),
      offset(name), XtRString, "default"},
#undef offset
};

```

该定义将允许 URLWidget 类的实现正确地对 XtNURLResource 资源进行修改。下面的定义是从 Template.c 程序拷贝过来的, 只是在绑定程序文件中所定义的 Initialize 和 ReDraw 函数域有所不同:

```

URLClassRec urlClassRec= {
    { /* core fields */

```



```

    /* superclass          */ (WidgetClass) &widgetClassRec,
    /* class_name           */ "URL"
    /* widget_size          */ sizeof(URLRec),
    /* class_initialize     */ NULL,
    /* class_part_initialize */ NULL,
    /* class_initialized    */ FALSE,
    /* initialize           */ Initialize, // CHANGED
    /* initialize_hook      */ NULL,
    /* realize              */ XtInheritRealize,
    /* actions              */ NULL, // actions,
    /* num_actions          */ 0, // XtNumber(actions),
    /* resources            */ resources,
    /* num_resources        */ XtNumber(resources),
    /* xrm_class            */ NULLQUARK,
    /* compress_motion      */ TRUE,
    /* compress_exposure    */ TRUE,
    /* compress_enterleave  */ TRUE,
    /* visible_interest     */ FALSE,
    /* destroy              */ NULL,
    /* resize               */ NULL,
    /* expose               */ ReDraw, //CHANGED
    /* set_values            */ NULL,
    /* set_values_hook      */ NULL,
    /* set_values_almost    */ XtInheritSetValuesAlmost,
    /* get_values_hook      */ NULL,
    /* accept_focus         */ NULL,
    /* version              */ XtVersion,
    /* callback_private     */ NULL,
    /* tm_table             */ NULL, // translations,
    /* query_geometry       */ XtInheritQueryGeometry,
    /* display_accelerator  */ XtInheritDisplayAccelerator,
    /* extension            */ NULL
},
{ /* url fields */
    /* empty      */ 0
}
};

```

在核心窗口部件类的 `Initialize` 函数执行完与这个新的 `URLWidget` 类有关的设置工作后，调用函数 `Initialize`：

```

static void Initialize(URLWidget request, URLWidget new) {
    XtGCMask valueMask;
    XGCValues values;
    printf("name = %s\n", new->url.name);
    // Get the URL data here:
    new->url.data = fetch_url(new->url.name);
}

```

```

    valueMask = GCForeground | GCBackground;
    values.foreground = BlackPixel(XtDisplay(new), 0);
    values.background = WhitePixel(XtDisplay(new), 0);
    new->url.gc = XtGetGC((Widget)new, valueMask, &values);
}

```

函数 `Initialize` 使用 `fetch_url` 函数来获得一个远端 Web 服务器的数据并为该窗口部件设置图形上下文 (graphic context, GC)。注意, 数据是通过使用窗口部件的 `url` 域进行访问的。函数 `clean_text` 很简单, 用来在从远端 Web 服务器的原始数据中提取出一行文本之后, 从这些文本中删除控制字符:

```

static char buf[50000];
static char buf2[50000];
char * clean_text(char *dirty_text) {
    int i, count = 0;
    int len = strlen(dirty_text);
    for (i=0; i<len; i++) {
        if (dirty_text[i] > 30) buf2[count++] = dirty_text[i];
    }
    buf2[count] = 0;
    return &(buf2[0]);
}

```

函数 `ReDraw` 在窗口部件初始化之后调用; 每当窗口部件暴露或发生大小变化时, `ReDraw` 函数都要被调用。`ReDraw` 只是删除每一行由换行符分割的文本中的任何控制字符, 并调用 `XDrawString` 函数在窗口部件的正确位置重写每行文本。在第 25 章中已经看过 `XDrawString` 函数的使用。该例子中不使用字体属性来决定每行文本的高度; 它假设 12 个像素高的一行足够用于放置一个文本行。

```

static void ReDraw(URLWidget w, XEvent *event, Region region) {
    //printf("in ReDraw, text is:\n%s\n", w->url.data);
    char *sp1, *sp2, *sp3;
    int len, y = 20;
    sp1 = &(buf[0]);
    sprintf(sp1, "%s", w->url.data);
    len = strlen(sp1);
    while (1) {
        sp2 = strstr(sp1, "\n");
        if (sp2 != NULL) {
            // keep going...
            *sp2 = '\0';
            sp3 = clean_text(sp1);
            XDrawString(XtDisplay(w), XtWindow(w), w->url.gc, 10, y,
                       sp3, strlen(sp3));
            y += 12;
            sp1 = sp2 + 1;
        } else {

```

```

        // time to stop...
        sp3 = clean_text(sp1);
        XDrawString(XtDisplay(w), XtWindow(w), w->url.gc, 10, y,
                    sp3, strlen(sp3));
        break;
    }
    // check to avoid running past data:
    if (sp1 >= &(buf[0]) + len) break;
}
}

```

URLWidget 的实现实际上非常简单。大部分工作在于严格地遵守用于编写窗口部件的 X 工具包协议。原则上, URLWidget 可以在任何 X 应用程序中使用, 也包括使用 Motif 窗口部件的应用程序。

26.3.5 测试 URLWidget

如果你还没有这样做, 请进入 src/X/URLWidget 目录并键入

```

make
./test

```

来编译并运行 test.c 程序。

这个测试程序很简单。使用窗口部件比编写代码来实现它要容易得多。在程序开始既包含了标准的 X11 头文件, 也包含了 URLWidget 的公共头文件:

```

#include <X11/StringDefs.h>
#include <X11/Intrinsic.h>
#include "URL.h"

```

主函数 main 定义了两个窗口部件 (Top_Level 和 URL), 并创建了 Top_Level 窗口部件:

```

Widget top_level, url;
top_level = XtInitialize(argv[0], "urltest", NULL, 0, &argc, argv);

```

在创建 URL 窗口部件之前, 需要为 URL 窗口部件设置宽度、高度和 URLResource 资源:

```

Arg args[3];
XtSetArg(args[0], XtNwidth, 520);
XtSetArg(args[1], XtNheight, 580);
XtSetArg(args[2], XtNURLResource, "www.knowledgebooks.com");
url=XtCreateManagedWidget("url", urlWidgetClass, top_level,
    args, 3);

```

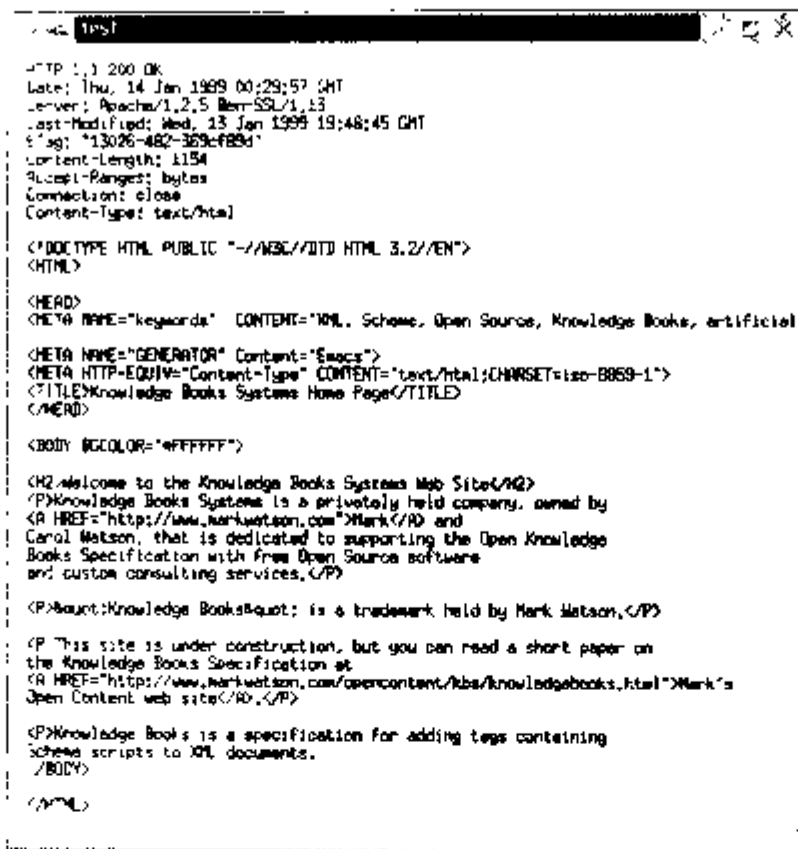
最后, 将 Top-Level 窗口部件变为可见的, 并处理 X 事件:

```

XtRealizeWidget(top_level);
XtMainLoop();

```

图 26.8 显示了程序 test.c 的运行情况。



```

HTTP/1.1 200 OK
Date: Thu, 14 Jan 1999 00:29:57 GMT
Server: Apache/1.2.5 Bern-SSE/1.1.3
Last-Modified: Wed, 13 Jan 1999 19:48:45 GMT
Etag: "13026-482-369cf89d"
Content-Length: 1154
Accept-Ranges: bytes
Connection: close
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
<HEAD>
<META NAME="keywords" CONTENT="XML, Scheme, Open Source, Knowledge Books, artificial">
<META NAME="GENERATOR" Content="Emacs">
<META HTTP-EQUIV="Content-Type" CONTENT="text/html;CHARSET=iso-8859-1">
<TITLE>Knowledge Books Systems Home Page</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<H2>Welcome to the Knowledge Books Systems Web Site</H2>
<P>Knowledge Books Systems is a privately held company, owned by
<A HREF="http://www.markwatson.com">Mark Watson</A> and
Carol Watson, that is dedicated to supporting the Open Knowledge
Books Specification with free Open Source software
and custom consulting services.</P>
<P>Knowledge Books is a trademark held by Mark Watson.</P>
<P>This site is under construction, but you can read a short paper on
the Knowledge Books Specification at
<A HREF="http://www.markwatson.com/opencontent/kbs/knowledgebooks.html">Mark's
Open Content web site</A>.</P>
<P>Knowledge Books is a specification for adding tags containing
scheme scripts to XML documents.
</BODY>
</HTML>

```

图 26.8 测试 URLWidget

26.4 在 C++程序中使用 Athena 和 Motif

当我在 20 世纪 80 年代后期初次使用 C++编写程序时，大体上同一时期我也在学习 X 窗口编程。当时的坏消息是 X 工具包和 Athena 窗口部件并不是非常的“C++友好”，因此我花费了大量的时间来将窗口部件和 C 程序进行结合。

好，现在的好消息是 X11 库函数、头文件等等都可以毫不费力的使用 C++语言工作。如果你看一下 src/X/list_C++目录，你将发现两个源文件：

athena.cpp——这是 src/X/Athena/list.c 文件经重命名得到的

motif.cpp——这是 src/X/Motif/list.c 文件经重命名得到的

除了一个关于 printf 的编译器警告外，这些例子都可以用 C++编译器进行编译并运行良好。自己试试，进入 src/X/list_C++目录并键入：

```

make
./athena
./motif

```

因为 C++编译器比 C 编译器提供了更好的编译时错误检查功能，我建议使用 C++来代替 C，即使你不使用 C++的面向对象特性（例如，没有类的定义）。

26.5 使用封装 Athena 窗口部件的一个 C++ 类库

在这一节，将学习一个非常简单的 C++ 库，它封装了表 26.6 中的 Athena 窗口部件：

表 26.6 被封装的 Athena 窗口部件

窗口部件	描述
Paned	封装在 C++ 类 PaneWindow（它也产生了一种高层应用的窗口部件）中
Labeled	封装在 C++ 类 Label 中
Command Button	封装在 C++ 类 Button 中
AsciiText	封装在 C++ 类 Text 中

出于该练习的某种目的，将首先展示一个简单的测试程序，它绝对不包含任何（明显的）X Windows 代码。示例程序 test_it.cpp 存放在 src/X/C++ 目录下，并在下文中列出。

iostream.h 定义了标准的 C++ 输入/输出。include 文件 PaneWindow.h、Label.h、Button.h 和 Text.h 定义了包含响应 Athena 窗口部件的 C++ 类：

```
#include <iostream.h>
#include "PaneWindow.hxx"
#include "Label.hxx"
#include "Button.hxx"
#include "Text.hxx"
```

我们将为每一个类创建一个对象。Button 将有一个用于打印文本对象内容的回调函数。我们令 Text 对象是全局对象，从而回调函数 button_cb 可以对其进行访问：

```
Text *text;
```

回调函数 button_cb 使用 Text 类的公共成员函数 getText 来打印文本对象中出现的任何文本数据：

```
void button_cb() {
    cout << "Text is:\n" << text->getText() << "\n";
}
```

主函数 main 非常简单。我们为每一个封装类构造一个实例，使用 PaneWindow 类的公共成员函数 addComponent 来添加 Label、Button 和 Text 对象到 PaneWindow 对象中，然后调用 PaneWindow 类的公共成员函数 run 来处理事件。注意，Button 类的构造函数接受一个 C 函数作为它的第二个参数；每当鼠标指针发生了单击事件，该函数将被调用。

```
void main() {
    PaneWindow pw;
    Label label("this is a test");
    Button button("Get text", button_cb);
    text = new Text(300, 120);
    pw.addComponent(&label);
    pw.addComponent(&button);
}
```

```

        pw.addComponent(&text);
        pw.run();
    }

```

图 26.9 显示了 test_it.cpp 程序的执行情况。

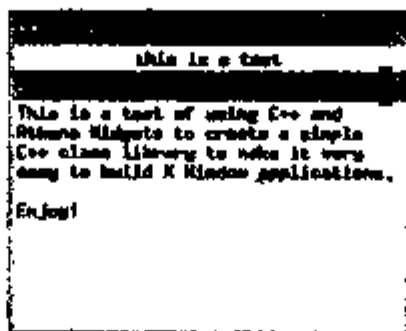


图 26.9 测试封装 Athena 窗口部件的 C++ 类库

PaneWindow 类在类库中做了大多数的的工作。该类负责添加子部件和处理 X 事件的工作。Label、Button 和 Text 类都是从基类 Component 中派生的，Component 类有一个重要的虚函数 setup。这样设计该类库的原因是 PaneWindow 类可以仅保持一个指向从 Component 类派生的类实例的指针数组即可。PaneWindow 类的公共成员函数 run 通过为每一个已加入 PaneWindow 对象的子部件调用成员函数 setup 来创建所有它所包含的部件，然后 run 函数在部件内部处理所有的 X 事件。

该类库的实现也很简单，你也应该发现为应用程序向该库中加入其他的窗口部件是很容易的，惟一复杂的是关于处理 C 语言的回调函数。不久将在 Button 类的实现中看到它是如何完成的。

26.5.1 Component 类

Component 类是 Label、Button 和 Text 类的基类。如果你想封装其他的窗口部件到一个 C++ 类中并将它们加入到类库中，你的新类也应该从 Component 类派生而来。该类的头文件非常简单。Component.hxx 文件的全部内容都“封装”在一个 #ifndef 预处理语句中，这对于头文件来说是很典型的。

```

#ifndef Component_hxx
#define Component_hxx

```

#ifndef 保证该文件将不会在任何编译的代码中被包含两次。我们还需要标准的 X 工具包 include 文件来定义 Top-Level 应用程序窗口部件的类型：

```

#include <X11/Intrinsic.h>

```

类的定义非常简单。类的构造函数几乎什么也不做，而虚函数 setup 必须在派生类中被重载，因为它是一个纯虚函数——该成员函数被置为 0。公共成员函数 getWidget 只是返回了私有变量 my_widget 的值。

```

class Component {
public:
    Component();

```

```

    virtual void setup(Widget parent) = 0;
    Widget getWidget() { return my_widget; }
protected:
    Widget my_widget;
};

```

文件 `Component.cxx` 中 `Component` 类的实现也很简单:

```

// Component.cpp
//
#include <X11/Intrinsic.h>
#include "Component.hxx"

Component::Component() {
    my_widget = (Widget) NULL;
}

```

`Component` 类的构造函数设置私有变量 `my_widget` 为 `NULL`, 实际上这并不是严格需要的。

26.5.2 PaneWindow 类

在看 `PaneWindow` 类之前先来看 `Component` 类的定义这一点很重要, 因为一个 `PaneWindow` 对象保留了一个指向从 `Component` 类派生出来的对象的指针数组。`PaneWindow` 类的头文件是 `PaneWindow.h`, 其中需要 4 个 `include` 文件:

```

#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xaw/Paned.h>
#include "Component.hxx"

```

前 3 个 `include` 文件是 X Windows 定义所需要的; 而第 4 个 `include` 文件是 `Component` 类的头文件。`PaneWindow` 类有 3 个公共成员函数, 如表 26.7 所示。

表 26.7 `PaneWindow` 类的 3 个公共成员函数

成员函数	描述
<code>PaneWindow</code>	<code>PaneWindow</code> 类的构造函数
<code>AddComponent(Component *comp)</code>	添加其他的部件到窗格窗口中
<code>run</code>	创建所有被包含的部件并进行事件处理

`PaneWindow` 类的私有数据包含了与 X Windows 紧密相关的数据, 使用该类库的用户可以忽略这些数据。还有私有数据用来存储最多 10 个指向属于 `Component` 类的任何派生类对象的指针。

```

class PaneWindow {
public:
    PaneWindow();
    void addComponent(Component * comp);

```

```

    void run();
private:
    Widget top_level;
    Widget pane;
    Component * components[10];
    int num_components;
    XtAppContext application_context;
};

```

文件 `PaneWindow.cxx` 中 `PaneWindow` 类的实现需要 3 个 `include` 文件；前两个是与 X 相关的，而第 3 个是类定义的头文件：

```

#include <X11/Intrinsic.h>
#include <X11/Xaw/Paned.h>
#include "PaneWindow.hxx"

```

下面的 X 资源用于定义可能被添加到一个窗格窗口的部件成分：

```

static String app_resources[] = {
    "**Text*editType: exit",
    "**Text*autoFill: on",
    "**Text*scrollVertical: whenNeeded",
    "**Text*scrollHorizontal: whenNeeded",
    "**Text*preferredPaneSize: 300",
    NULL,
};

```

如果你创建新的 `Component` 类的子类来封装其他类型的窗口部件，你可能想为这些窗口部件类型添加默认的资源到 `app_resources` 数组中。`PaneWindow` 类的构造函数创建一个 Top-Level 应用程序窗口部件和一个 Athena 窗格窗口部件：

```

PaneWindow::PaneWindow() {
    int argc = 0;
    char ** argv = NULL;
    top_level = XtAppInitialize(&application_context, "top_level",
                               NULL, 0, &argc, argv, app_resources,
                               NULL, 0);

    num_components = 0;
    pane = XtVaCreateManagedWidget("paned", panedWidgetClass,
                                    top_level, NULL);
}

```

成员函数 `run` 创建被添加到 `PaneWindow` 对象的组件，然后处理 X 事件。类变量 `num_components` 是一个用来计算通过调用 `addComponent` 方法添加到该对象中的组件个数的计数器：

```

void PaneWindow::run() {
    // Start by adding all registered components:
    for (int i=0; i<num_components; i++) {

```



```

        components[i]->setup(pane);
        XtManageChild(components[i]->getWidget());
    }
    XtRealizeWidget(top_level);
    XtAppMainLoop(application_context);
}

```

成员函数 `addComponent` 用来添加组件到一个 `PaneWindow` 对象中:

```

void PaneWindow::addComponent(Component * comp) {
    components[num_components++] = comp;
}

```

26.5.3 Label 类

`Label` 类是从 `Component` 类派生出来的。类的头文件 `Label.hxx` 仅需要一个头文件, 该头文件是其基类的定义文件:

```
#include "Component.hxx"
```

类的定义很简单。构造函数接受 3 个参数, 类具有私有数据, 从而在 `PaneWindow` 对象的 `run` 方法中进行调用时, 成员函数 `setup` 可以适当地创建标签:

```

class Label : public Component {
public:
    Label(char * label = "test label", int width=100, int height=20);
    void setup(Widget parent);
private:
    char * my_label;
    int my_width;
    int my_height;
};

```

该类的实现同样很直观。文件 `Label.cxx` 需要四个 `include` 文件, 其中三个是与 X Windows 有关的; 剩下的一个用来定义 `Label` 类:

```

#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xaw/Label.h>
#include "Label.hxx"

```

类的构造函数通过调用 `setup` 方法保存了它的三个参数以备后面使用:

```

Label::Label(char *label, int width, int height) {
    my_label = label;
    my_width = width;
    my_height = height;
}

```

`setup` 方法被 `PaneWindow` 对象的成员函数 `run` 调用; `setup` 和 `PaneWindow` 对象的构造函数所创建的 Athena 窗格窗口部件一起调用。因此所有添加的窗口部件将 Athena 窗格窗

口部件作为它们公共的父亲:

```
void Label::setup(Widget parent) {
    Arg args[2];
    XtSetArg(args[0], XtNwidth, my_width);
    XtSetArg(args[1], XtNheight, my_height);
    my_widget = XtCreateManagedWidget(my_label, labelWidgetClass,
                                       parent, args, 2);
}
```

26.5.4 Button 类

Button 类比 Label 要有趣得多, 因为它必须处理回调函数。头文件 Button.hxx 需要两个 include 文件: 一个是用来引入 Athena 命令行窗口部件, 一个是用来定义 C++ 类 Component:

```
#include <X11/Xaw/Command.h>
#include "Component.hxx"
```

头文件将回调函数的类型定义为指向一个既没有参数又没有返回值的函数的指针:

```
typedef void (*button_callback) (void);
```

类的定义如下所示:

```
class Button : public Component {
public:
    Button(char * label = "test label", button_callback cb = 0,
           int width=130, int height=30);
    void setup(Widget parent);
    button_callback my_cb;
private:
    char * my_label;
    int my_width;
    int my_height;
};
```

类的实现文件 Button.cxx 需要四个 include 文件: 三个用于 X Windows; 一个用于定义 C++ 的 Button 类:

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xaw/Command.h>
#include "Button.hxx"
```

Button 类的所有实例都使用一个静态的 C 回调函数。为了使 Button 对象的回调函数可以被调用, 该函数接受一个指向 Button 对象的指针作为参数。这将允许具有不同回调函数的多个 Button 类实例在一个程序中正确地工作:

```
static void callback(Widget w, caddr_t data1, caddr_t data2) {
    Button * b = (Button *)data1;
    if (b != 0) b->my_cb();
}
```

类的构造函数将它的 4 个参数值存储在其私有数据区:

```
Button::Button(char *label, button_callback cb,
               int width, int height) {
    my_label = label;
    my_width = width;
    my_height = height;
    my_cb = cb;
}
```

成员函数 **setup** 创建 button 窗口部件并设置由该类的所有实例所共享的回调函数:

```
void Button::setup(Widget parent) {
    Arg args[2];
    XtSetArg(args[0], XtNwidth, my_width);
    XtSetArg(args[1], XtNheight, my_height);
    my_widget = XtCreateManagedWidget(my_label, commandWidgetClass,
                                      parent, args, 2);
    XtAddCallback(getWidget(), XtNcallback, callback, (XtPointer)
                  this);
}
```

26.5.5 Text 类

Text 类封装了 Athena 的 AsciiText Top-Level 应用程序窗口部件。类定义文件 Text.hxx 需要两个 include 文件: 一个用于标准的 X Windows 定义, 一个用于 C++ Component 类的定义:

```
#include <X11/Intrinsic.h>
#include "Component.hxx"
```

类定义中定义了 4 个公共成员函数, 如表 26.8 所示。

表 26.8 Text 类定义中定义的 4 个公共成员函数

成员函数	描述
Text(int width,int height)	类构造函数
setup(Widget parent)	创建 Text 窗口部件
char *getText	从 Text 窗口部件中获得文本
eraseText	删除 Text 窗口部件中的所有文本

该类定义了用来存储类构造函数两个参数的私有数据。

```

class Text : public Component {
public:
    Text(int width=130, int height=30);
    void setup(Widget parent);
    char *getText();
    void eraseText();
private:
    int my_width;
    int my_height;
};

```

文件 `Text.cxx` 中的实现相当简单。这里需要 4 个 `include` 文件：3 个用于 X Windows 的定义，1 个用于 C++ `Text` 类的定义：

```

#include <X11/Intrinsic.h >
#include <X11/StringDefs.h >
#include <X11/Xaw/AsciiText.h >
#include "Text.hxx"

```

类构造函数将它的两个参数通过 `setup` 方法保存在私有数据区，以备将来使用：

```

Text::Text(int width, int height) {
    my_width = width;
    my_height = height;
}

```

`setup` 方法用来创建 Athena 的 Top-Level 应用程序窗口部件：

```

void Text::setup(Widget parent) {
    Arg args[2];
    XtSetArg(args[0], XtNwidth, my_width);
    XtSetArg(args[1], XtNheight, my_height);
    my_widget = XtVaCreateManagedWidget("text",
        asciiTextWidgetClass, parent,
        XtNtype, XawAsciiString,
        XtNstring, " ", args, 2, NULL);
}

```

`getText` 方法返回用户输入到 Athena 的 `AsciiText` Top-Level 应用程序窗口部件中的文本。X 工具包函数 `XtVaGetValues` 用于获取被键入窗口部件的文本字符串数据的地址：

```

char * Text::getText() {
    Widget w = getWidget();
    String str;
    XtVaGetValues(w,
        XtNstring, &str,
        NULL);
    return str;
}

```

eraseText 方法从 Athena 的 AsciiText Top-Level 应用程序窗口部件中删除所有文本。这里，我们也可以使用 X 工具包函数 XtSetValues 来实现同样的功能。

```
void Text::eraseText() {  
    Widget w = getWidget();  
    XtVaSetValues(w,  
                  XtNstring, "",  
                  NULL);  
}
```

本节所开发的简单的 C++ 类库提供了用于将其他窗口部件类型封装到 C++ 类库中的技术。

26.6 小 结

在本章中，看到了怎样使用 Athena 和 Motif 窗口部件。这两种窗口部件各有所长。还了解了使用 C++ 语言编写基于 Athena 和基于 Motif 的程序的技术。

Athena 和 Motif 在一定程度上是一种历史遗留库，它们正在被更新的、更好的窗口部件库所取代。对于 Linux 编程来说，更是如此。大多数用于 Linux 的新的、现代的 X Windows 应用都使用了 GTK+ 或者 Qt 作为它们的窗口部件。但是，如果想为其他 UNIX 系统开发程序，特别是如果你准备进行商业开发时，将发现 Athena 和 Motif 是最常用的窗口部件。你甚至会发现 Qt 库提供了对 Motif 的部分支持——如果它不算别的什么，但至少具有 Motif 的风格模式。

第 27 章 使用 GTK+进行 GUI 编程

在 Linux 和其他版本的 UNIX 上, Gimp Tool Kit(GTK+)广泛地用于编写 X Windows 应用。为了有助于保持可移植性和软件维护, GTK+由表 27.1 中的 3 个库所组成, 在一定程度上, 你可以各自独立地使用它们中的每一个。

表 27.1 GTK+包含的 3 个库

库	描述
GLib	G 库。为链接列表、散列表、字符串工具等提供 C 函数。
GDK	代表 GTK+ Drawing Kit (GTK+绘图包)。它是在 Xlib 之上的一层库。GTK+所有的窗口创建和图形调用都要通过 GDK, 而不是直接面向 Xlib。它也是把 GTK+移植到一个不同的操作系统时惟一需要重写的库。GTK+在 Windows 上还有一个移植版本。
GTK	代表 Gimp Tool Kit。一种高级的窗口部件集。

正如从这个列表中所看到的那样, 窗口部件库和整个工具都叫作 Gimp Tool Kit。在本章里, 我们将把整个软件包称为 GTK+, 而把其中的库和函数称为 GTK。GTK+中的加号用来表示它是下一代 GTK, 而且实现了类似面向对象的特性。

GTK+项目有它自己的 Web 站点 (<http://www.gtk.org>), 可以从这个站点下载最新版的 GTK+。你还可以找到出色的 FAQ, 以及 Ian Main 和 Tony Gale 编写的不错的 GTK 教程。在这个站点上还有其他 GTK+资源, 比如绑定除了 C 之外的其他编程语言。

在本章中将介绍 GTK+并且编写一个简单的用于显示任何 XML 文档树状结构的 GTK+应用程序。扩展标记语言 (eXtensible Markup Language, XML) 是用于存储和传送数据的新标准, 所以这个短小的示例程序应该很有用处。即使本章的内容相对充分, 但还是建议你也阅读 GTK+的教程。

GTK+是用于编写 GUI 应用的一种易用的高层工具包。编写 GTK+的目的是为了支持 GIMP 这一图形编辑程序。GTK+最初是由 Peter Mattis、Spencer Kimball 和 Josh MacDonald 开发编写的。

GTK+提供了一个包含丰富的数据类型集和工具函数的工具包。对 GTK+的完整描述需要一本专门的书籍, 这超出了本书的范围。但是阅读本章的内容可以帮助初学者入门。要了解更多信息, 请参阅网上的教程和包括在标准 GTK+发布版本中 examples 目录下的例程。

GTK+库代码和例程的代码均包含下面的版权信息。在本章的示例程序中, 照搬了这种风格 (例如, 变量的命名规则) 和部分代码行, 所以我认为本章所有的示例程序都是派生的, 因此也应该遵守下列版权声明:

```
/* GTK - The GIMP Toolkit
 * Copyright (C) 1995-1997 Peter Mattis, Spencer Kimball and Josh
 * MacDonald
```

```

*
* This library is free software; you can redistribute it and /or
* modify it under the terms of the GNU Library General Public
* License as published by the Free Software Foundation; either
* version 2 of the License, or (at your option) any later version.
*
* This library is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
* Library General Public License for more details.
*
* You should have received a copy of the GNU Library General Public
* License along with this library; if not, write to the
* Free Software Foundation, Inc., 59 Temple Place - Suite 330,
* Boston, MA 02111-1307, USA.
*/

```

27.1 GTK+简介

在第26章“Athena、Motif和LessTif窗口部件”中使用了Athena和Motif的窗口部件。GTK+具有自己的窗口部件类型；这种类型的C数据结构称为GtkWidget。使用GTK+创建的窗口以及任何显示部件均可以通过类型为GtkWidget的变量来引用。尽管GTK+是用C写成的，但是GTK+具有很强的面向对象风格。一个GtkWidget对象具有这样的功能：封装数据、维护回调函数的引用等等。

GtkWidget数据类型的定义（在gtkwidget.h中）如下：

```

struct _GtkWidget {
{
    GObject object; // core GTK object definition
    guint16 private_flags; // private storage
    guint8 state; // one of 5 allowed widget states
    guint8 saved_state; // previous widget state
    gchar *name; // widget's name
    GtkStyle *style; // style definition
    GtkRequisition requisition; // requested widget size
    GtkAllocation allocation; // actual widget size
    GdkWindow *window; // top-level window
    GtkWidget *parent; // parent widget
};

```

通过在结构定义中将GtkWidget对象定义为第一项，GTK的窗口部件“继承”了GtkWidget。例如，通过GtkMisc结构可以定义一个GtkLabel窗口部件，只需添加标签的文本、宽度、类型和指明是否自动换行的标志数据即可，如下所示：

```

struct _GtkLabel {
    GtkMisc misc;
    gchar    *label;
    GdkWChar  *label_wc;
    gchar    *pattern;
    GtkLabelWord *words;
    guint    max_width : 16;
    guint    jtype : 2;
    gboolean wrap;
};

```

GtkMisc 结构定义从一个 GtkWidget 结构开始，在后面添加对齐和填充信息，如下所示：

```

Struct _GtkMisc {
    GtkWidget widget;
    gfloat xalign;
    gfloat yalign;
    guint16 xpad;
    guint16 ypad;
};

```

这里要注意的重要一点是，尽管一个 GtkLabel 的结构建立在下列嵌套结构中，GtkWidget 结构的数据在分配给 GtkLabel 窗口部件的内存区域的起始处：

```

GtkLabel
    GtkMisc
        GtkWidget
            GObject

```

这意味着指向任何类型的 GtkWidget 的指针均可以安全地强制转换为 GtkWidget 指针类型 (GtkWidget *)，GtkWidget 结构的所有字段均可以直接访问。因此，即使 GTK 是用 C 语言实现的，它也不支持私有数据，但是使用了许多源自于面向对象理论的思想 and 概念。

27.1.1 在 GTK+中处理事件

在第 26 章已了解到，对于 Athena 和 Motif 窗口部件编程，编写 GUI 应用程序的主要任务包括两点：创建窗口部件；编写代码以处理 callback 函数中的事件。GTK 也使用回调函数处理程序中的事件，但在 GTK 中它们称为“信号处理函数”。其技巧和第 25、26 章的中的 X Windows 编程示例中的技巧非常相像，GTK 回调函数或信号处理程序在文件 gtkwidget.h 中定义，如下所示：

```

typedef void (*GtkCallback) (GtkWidget *widget, gpointer data);

```

我们在上一节看到了 GtkWidget 的定义。gpoint 是一种抽象指针，可以引用任意类型的数据。

正如要在下一节的例子中看到的那样，GTK 函数 gtk_main 使用 gtk_signal_connect 函数将事件与 GTK 窗口部件绑定，通过这种方式处理所有注册事件。函数原型（在文件

gtksignal.h 中定义) 定义如下:

```
guint gtk_signal_connect(GtkObject *object,
                        const gchar *name,
                        GtkSignalFunc func,
                        gpointer func_data);
```

在 gtksignal.h 中定义了 34 个不同的函数, 用于注册或解除注册信号处理函数。然而, 在本章的例子中使用 `gtk_signal_connect` 就够了。函数原型如下:

```
guint gtk_signal_connect (GtkObject *object,
                        const gchar *name,
                        GtkSignalFunc func,
                        gpointer func_data);
```

第一个参数是指向某个 `GtkObject` 的指针。然而在实际中传递的是 `GtkWidget` 对象的地址。`GtkWidget` 在其结构定义的起始处包含一个 `GtkObject`, 因此将一个 `GtkWidget` 指针强制转换成 `GtkObject` 指针是安全的。`gtk_signal_connect` 的第二个参数是事件类型的名称。最常用的 GTK 事件类型名称如表 27.2 所示。

表 27.2 最常用的 GTK 事件类型

GTK 事件类型	描述
clicked	用于按钮窗口部件
destroy	用于窗口部件
value_changed	用于任意类型的 <code>GtkObject</code>
toggled	用于任意类型的 <code>GtkObject</code>
activate	用于任意类型的 <code>GtkObject</code>
button_press_event	用于任意类型的窗口部件
select_row	用于列表窗口部件
select_child	用于树窗口部件
unselect_child	用于树窗口部件
select	用于项目窗口部件 (例如, 可以添加到树中的窗口部件)
deselect	用于项目窗口部件
expose_event	当窗口部件第一次创建或出现时发生
configure_event	当窗口部件第一次创建或调整大小时发生

27.1.2 使用 GTK+的简短示例程序

在 `src/GTK` 目录下的 `simple.c` 文件中包含了一个简单的 GTK 应用程序, 它将单个 GTK 按钮窗口部件放在某个窗口中, 并将一个 `callback` 函数 (一个信号处理函数) 连接到按钮上, 在每次单击按钮时调用本地函数 `do_click`。这个文件使用了一个 `include` 文件, 此 `include` 文件对所有的 GTK+应用程序都是必须的:

```
#include <gtk/gtk.h>
```

回调函数 `do_click` 的类型为 `GtkCallback`，每次单击按钮窗口部件时即调用此函数。我们会在主函数定义中看到，一个整型计数器变量的地址作为此回调（或称信号处理函数）函数的程序数据而传递。无论 `gtk_main` 中的 GTK 事件处理代码何时调用此函数，它都会将此整型变量的地址作为第二个参数传递。我们在函数 `do_click` 中所做的第一件事是将这个抽象数据指针强制转换为整数指针类型（`int *`）。现在 `do_click` 函数可以更改 `main` 函数中声明的计数器变量值。在 GTK+ 程序中，可以为回调程序数据指定任何类型的数据。

```
void do_click(GtkWidget *widget, gpointer data)
{
    int * count = (int *) data;
    *count += 1;
    printf("Button clicked %d times (5 to quit) \n", *count);
    if (*count > 4) gtk_main_quit();
}
```

`simple.c` 文件中定义的 `main` 函数很简单，它使用了表 27.3 中的 8 个 GTK 实用工具函数。

表 27.3 `simple.c` 文件中使用的 8 个 GTK 实用工具函数

函数	描述
<code>gtk_init</code>	将程序的命令行参数传递给 GTK 初始化代码。作为 GTK 选项处理的参数都从参数列表中移去。可用的命令行参数列表可在 http://www.gtk.org 的 GTK 教程中找到
<code>gtk_window_new</code>	创建一个新窗口。这个函数通常用于创建顶层应用程序窗口，在这个例子中就是这样
<code>gtk_container_border_width</code>	此函数可选，它用于设置添加到窗口的窗口部件周围的填充像素数目
<code>gtk_button_new_with_label</code>	用指定的标签创建新的按钮标签
<code>gtk_signal_connect</code>	上一节中我们已看到此函数如何将回调函数分配给指定类型事件对应的窗口部件
<code>gtk_container_add</code>	此函数用于将窗口部件添加到窗口或其他任何容器窗口部件中
<code>gtk_widget_show</code>	此函数使一个窗口部件可见，子窗口部件在父窗口部件之前可见
<code>gtk_main</code>	完成初始化，处理事件

程序清单 27.1 显示 `main` 函数的源代码，其后是对这些代码的讨论。

程序清单 27.1 `main` 函数，摘自 `simple.c`

```
int main(int argc, char *argv[]) {
    // Declare variables to reference both a
    // window widget and a button widget:
    GtkWidget *window, *button;
    // Use for demonstrating encapsulation of data(the address
    // of this variable will be passed to the button's callback
    // function):
```

```

    int count = 0;

    // Pass command line arguments to the GTK initialization function:
    gtk_init(&argc, &argv);

    // Create a new top-level window:
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

    // Change the default width around widgets from 0 to 5 pixels:
    gtk_container_border_width(GTK_CONTAINER(window), 5);

    button = gtk_button_new_with_label("Click to increment
                                     counter");

    // Connect the 'do_click' C callback function to the button widget.
    // We pass in the address of the variable 'count' so that the
    // callback function can access the value of count without having
    // to use global data:
    gtk_signal_connect(GTK_OBJECT(button), "clicked",
                      GTK_SIGNAL_FUNC(do_click), &count);

    // Add the button widget to the window widget:
    gtk_container_add(GTK_CONTAINER(window), button);

    // Make any widgets added to the window visible before
    // making the window itself visible:
    gtk_widget_show(button);
    gtk_widget_show(window);

    // Handle events:
    gtk_main();
}

```

main 函数创建两个 GtkWidget 对象指针：window 和 button。这些窗口部件指针用于引用顶层的窗口和一个按钮窗口部件。在调用 gtk_signal_connect 时，我们使用宏强制参数进行类型转换，将参数转换为正确的类型，如下所示：

```

    gtk_signal_connect(GTK_OBJECT(button), "clicked",
                      GTK_SIGNAL_FUNC(do_click), &count);

```

例如，在文件 gtktypeutils.h 中，GTK_SIGNAL_FUNC 是这样定义的：

```

#define GTK_SIGNAL_FUNC(f)    ((GtkSignalFunc) f)

```

27.1.3 各种 GTK 窗口部件

本章仅用了两个示例程序：上一小节的 simple.c 和 27.2 节中描述的 XMLviewer.c。因此，本章我们仅使用下列几种 GTK 窗口部件：

- Window
- Scrolled Window
- Button
- Tree

- Tree Item

目前可用的所有 GTK+ 窗口部件都汇集在 <http://www.gtk.org> 的在线教程的文档中；在这一节里，我们列出部分常用的 GTK 窗口部件，这些窗口部件并未全部在本章涉及，但它们很重要。

Adjustment 窗口部件是可由用户更改其值的控件，这种更改通常通过鼠标指针来完成。Tooltips 窗口部件是一块小的文本区，当鼠标悬停在某个窗口部件之上时，就会显示 Tooltips。对话框窗口部件可用于有模式与无模式两种对话框。文本窗口部件允许用户输入和编辑一行文本。File Selection 窗口部件使得用户能选择任意文件。

CList 窗口部件实现一个二维的网格。在 GTK+ 应用程序中使用“菜单工厂”功能函数可以轻松地创建菜单。文本窗口部件允许用户编辑多行文本表单。

鼓励用户编译 GTK+ 发布版 examples 目录的子目录下的示例程序。图 27.1 显示了 4 个示例程序：notebook, dial, file selection 和 button。大约 10 分钟就可以编译并运行 GTK+ 中包括的所有示例程序。花这点时间是值得的，因为你不仅可以熟悉 GTK 窗口部件的外观和风格，还可以看到每个示例程序的源代码文件，这些文件都很短。

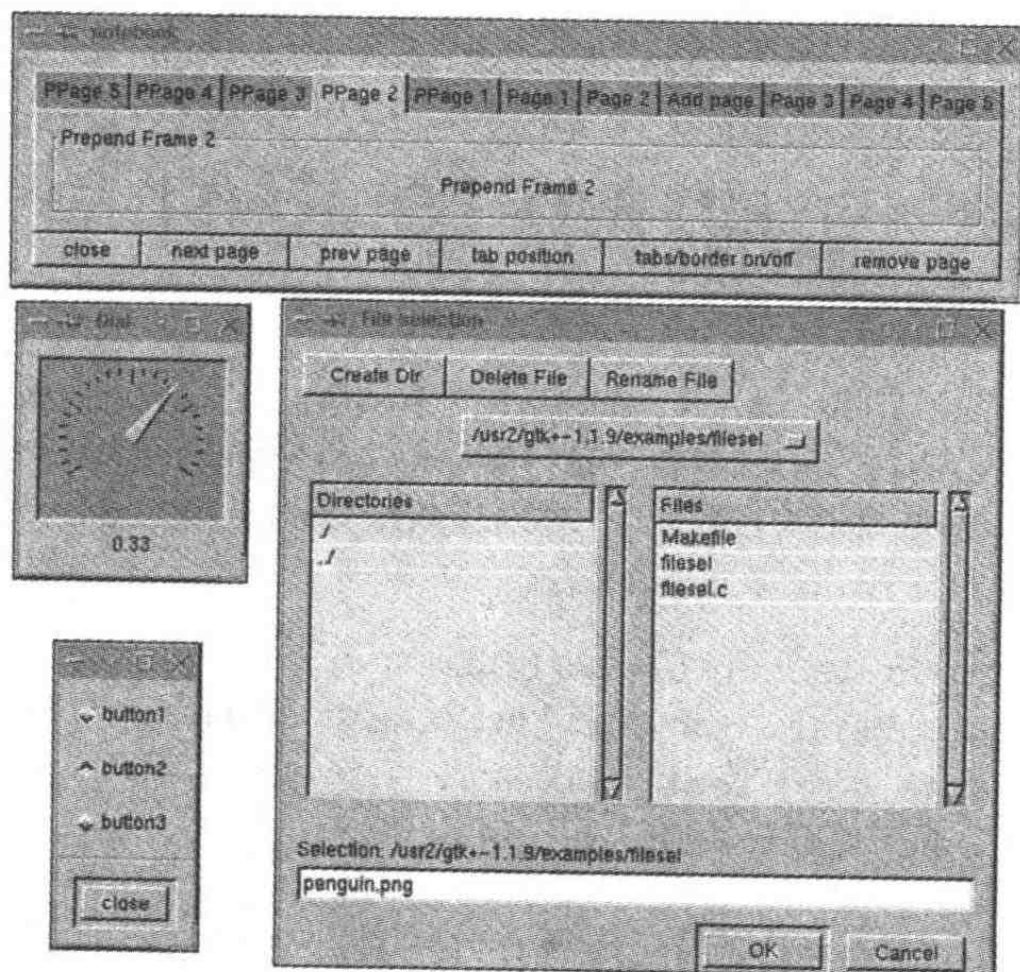


图 27.1 标准 GTK 发布版中包含的 4 个示例程序

27.1.4 GTK 容器窗口部件

尽管 GTK+ 中有许多种容器窗口部件（请参阅在线教程），在这一节里我们仅使用其中的一种：Scrolled Window 窗口部件。Scrolled Window 窗口部件用于 Window 窗口部件内部，提供一个虚拟的显示区域，这个潜在的区域比窗口的实际大小要大。稍后我们可以在示例程序 XMLviewer.c 中看到，Scrolled Window 窗口部件的默认操作是根据需要动态创建滚动栏（例如，窗口可以调整的更小，滚动栏也需要相应调整）。其他类型的容器窗口部件如表 27.4 所示。

表 27.4 其他类型容器窗口部件

容器窗口部件	描述
Notebook	一组有标签的选项卡，允许用户从某个窗口部件的若干浏览区或页中选择一个
Paned Window	将窗口区域分隔为两个浏览窗格。窗格间的分界线可由用户调整
Tool Bar	包含一行按钮，用于选择程序选项

如果你从 <http://www.gtk.org> 下载并安装了当前版本的 GTK+ 软件，则可以在 examples 目录下查找到使用了所有类型容器窗口部件的示例程序。

27.2 一个用于显示 XML 文件的 GTK+ 程序

在这一节里，我们开发了一个趣味性的示例程序，这个程序从标准输入读取 XML 文件，然后分析此文件，最后使用 GTK 的 Tree 窗口部件显示文件的树型结构。在你喜欢的编辑器中打开这段源代码是个不错的想法，这样在你阅读本章的时候可以很容易地进行参考。

例程使用了 James Clark 编写的 XML 分析器，它位于 src/GTK/expat 目录下。你可以在 Clark 的 Web 站点 (<http://www.jclark.com>) 上找到更新的版本，在这里还有许多有用的工具，可用于处理 XML 和 SGML 文档。

27.2.1 XML 简介

如果你在维护自己的网站，则可能对 HTML（超文本标记语言）已经很熟悉了。HTML 提供各种标记，指明 Web 文档的结构。一个 HTML 文件示例如下：

```
<HTML>
  <TITLE>This is a title</TITLE>
  <BODY>
    This is some test text
    for the page. <B>Cool</B>
  </BODY>
</HTML>
```

HTML 提供的预定义标记，如表 27.4 所示。

表 27.4 HTML 提供的预定义标记

标志	描述
TITLE	指明网页的标题
BODY	指明显示在网页上的文本
B	将文本的字体加粗

从示例中我们可以看到，标记类型符号“<>”标识了标记的开始，而“</>”标识了标记的结束。Web 浏览器知道如何识别合法的 HTML 标记。XML 看上去和 HTML 很类似，

但具有下列重要区别:

- 可以添加新的标记类型。
- XML 文档的所有部分都是区分大小写的。例如, <HTML>和<html>在 XML 中认为是不一样的。
- 每个打开标记必须有一个与之匹配的关闭标记。HTML 里没有这一要求。

程序清单 27.2 显示了一个简短的 XML 文件示例 test.xml, 此文件在本章的 src/GTK 目录下。

程序清单 27.2 XML 文件示例 test.xml

```
<?xml version="1.0"?>
<book>
  <title>A test title</title>
  <author>
    <last_name>Watson</last_name>
    <first_name>Mark</first_name>
  </author>
  <scheme>
    (define fff
      (lambda (x) (+ x x)))
    (define factorial
      (lambda (n)
        (if (= n 1)
            1
            (* n (factorial (- n 1))))))
  </scheme>
</book>
```

在这个 XML 文档中,我们使用了 book,title,author, last_name, first_name 和 scheme 等标记。第一行是标题行,向 XML 分析器指明此文件是 XML 1.0 兼容的。这个 test.xml 是一个“组织良好的”XML 文件,但它是“无效的”。一个有效的 XML 文档应该是组织良好的,同时还应有一个文档类型定义 (Document Type Definition, 即 DTD), 这个定义要么包含在文件中,要么在文件的开始处指明其参照之处。关于 DTD 的讨论超出了我们这里对 XML 的讨论范围,可以到下列网站查找详细信息:

<http://www.w3.org/XML/>
<http://www.software.ibm.com/xml/>

27.2.2 James Clark 的 XML 分析器 expat

有许多可用的 XML 分析器,这些分析器用不同的语言写成(例如 C, C++, Java, Python 和 Perl)。在本节里我们介绍一个用 C 写成的 XML 分析器,由 James Clark 先生开发,这个分析器的 Web 网址是 www.jclark.com/xml/expat.html。

可以在 CD-ROM 中的 src/GTK/expat 目录下找到这个分析器。我们会在下一节使用这个分析器编写 GTK+ 程序,显示 XML 文档的树型结构和内容。

src/GTK/expat/sample 目录下的 elements.c 文件显示基本分析器的用法。无论何时，只要分析器看到下列三个元素之一，就会调用使用 expat 分析器的应用程序所定义的回调函数：

- 某个标记
- 标记中的数据
- 某个结束标记

分析器对 XML 文件的树型结构进行深度优先的查找，遇到这些类型的元素时调用相应的回调函数。例如，示例程序 elements.c 在 test.xml 文件上生成下列输出：

```
# elements <test.xml
tag name: book
    tag name: title
        tag data: A test title
    tag name: author
        tag data:
            tag name: last_name
                tag data: Watson
            tag data:
                tag name: first_name
                    tag data: Mark
    tag name: scheme
        tag data:      (define fff
        tag data:      (lambda (x) (+ x x)))
        tag data:      (define factorial
        tag data:      (lambda (n)
        tag data:      (if (= n 1)
        tag data:      1
        tag data:      (* n (factorial (- n 1)))))
markw@colossus:/home/markw/MyDocs/LinuxBook/src/GTK/expat
/sample>
```

27.2.3 实现 GTK+ 的 XML 显示程序

已经介绍过一个简短的 XML 文件示例 (src/GTK 目录下的 test.xml)，我们将考察 XMLviewer 应用程序的具体实现。这里的文件 XMLviewer.c 部分派生于 James Clark 先生的示例程序 elements.c (在 CD-ROM 中的 src/GTK/expat/sample 目录下)。XMLviewer 程序需要下面 3 个 include 文件：

```
#include <stdio.h>
#include <gtk/gtk.h>
#include "xmlparse.h"
```

stdio.h 是必须的，因为需要从 stdin 读取 XML 输入文件，而 stdin 在 stdio.h 中定义。所有的 GTK+ 应用程序都需要包含 gtk.h。头文件 xmlparse.h 是使用 James Clark 的 XML 分

析器 expat 的标准头文件。

XMLviewer 应用程序创建一个 GTK Tree 窗口部件，并在树中的相应位置上为从标准输入中读入的 XML 文件中的每个 XML 元素（或标记）添加一个 GTK Tree Element 窗口部件。在应用程序中，GTK Tree 窗口部件处理打开与折叠子树的必要事件。然而，我们将 item_signal_callback 函数定义为对树视图中的鼠标选择事件进行处理的一个示例。item_signal_callback 的第一个参数是指针，指向 GTK 的 Tree Item 窗口部件，第二个参数则是信号的名称。由于为添加到树中的每个 Tree Item 窗口部件放置一个 GTK 标签窗口部件作为其子节点，变量 label 可以设置为此树项目的标签窗口部件的地址。GTK 实用工具函数 gtk_label_get 可用于获得构成标签的字符，然后即可打印出标签内容。item_signal_callback 的定义如下：

```
static void item_signal_callback(GtkWidget *item, gchar *signame)
{
    gchar *name;
    GtkLabel *label;
    label = GTK_LABEL(GTK_BIN(item)->child);
    /* Get the text of the label */
    gtk_label_get(label, &name);
    printf("signal name=%s, selected item name=%s\n", signame,
           name);
}
```

然后定义所需的数据，对 XML 元素的处理过程进行跟踪。这其中包括指向当前 GTK Tree 窗口部件和 Tree Item 窗口部件的指针。

```
GtkWidget *tree;
GtkWidget *item;
```

作为 XML 文档中的分析过程子树，我们使用 subtree[] 数组存放窗口部件指针：

```
#define MAX_DEPTH 10
GtkWidget *subtree[MAX_DEPTH];
```

XMLviewer.c 中的代码部分定义了 3 个回调函数，用于 expat 分析器。我们定义下列回调函数：

- **handleElementData**——调用此函数时需要使用打开与关闭标记之间包含的数据。例如，当处理元素“<author> Mark Watson </author>”时，标记间的数据就是“Mark Watson”。
- **startElement**——调用时需要使用首次被处理的标记名称。
- **endElement**——在调用 handleElementData 之后调用，需要使用标记的名称。

函数 handleElementData 由 expat 分析器调用，处理标记内的数据。函数原型如下：

```
void handleElementData(void *userData, const char * data, int len)
```


在本章里, 有关 GTK 的数据访问机制并无特殊之处, 下列代码指明创建新的 GTK Tree Item 窗口部件的方法:

```
int *depthPtr = userData;
GtkWidget *subitem;
cp = (char *)malloc(len + 1);
for (i=0; i<len; i++) cp[i] = data[i];
cp[len] = '\0';
subitem = gtk_tree_item_new_with_label(cp);
gtk_tree_append(GTK_TREE(subtree[*depthPtr]), subitem);
gtk_signal_connect(GTK_OBJECT(subitem), "select",
                   GTK_SIGNAL_FUNC(item_signal_callback),
                   "tag data select");
gtk_signal_connect(GTK_OBJECT(subitem), "deselect",
                   GTK_SIGNAL_FUNC(item_signal_callback),
                   "tag data deselect");
gtk_widget_show (subitem);
```

在这里, **expat** 分析器传递标记中的字符数据的地址; 有必要复制这些数据地址, 并使用这一副本创建 GTK 项目窗口部件。GTK 实用工具函数 `gtk_tree_item_new_with_label` 创建新的带标签的 Tree Item 窗口部件。新创建的 GTK Tree Item 窗口部件必须添加到 GTK Tree 窗口部件的适当位置上。这是通过下列函数调用完成的:

```
gtk_tree_append(GTK_TREE(subtree[*depthPtr]), subitem);
```

expat 分析器指明项目窗口部件在树中放置的深度。`GtkWidget` 指针数组 `subtree` 在 **expat** 的回调函数中填充。GTK Tree Item 窗口部件为 `select` 和 `deselect` 事件分别设置一个回调函数(或信号处理函数)。正如前面看到的那样。两个信号均由函数 `item_signal_callback` 处理。`handleElementData` 函数完成的最后一个操作是显示新创建的 GTK Tree Item 窗口部件。

`startElement` 函数由 **expat** 分析器调用, 对 XML 中的开始标记进行处理(例如“<author>”)。此函数的函数原型如下:

```
void startElement(void *userData, const char* name, const char
                  **atts)
```

标记的深度在参数 `userData` 中传递。此抽象指针将转换为整型指针, 指向的整数表示当前的 XML 树的标记深度:

```
int *depthPtr = userData;
*depthPtr += 1;
```

根据 XML 文档的分析树当前的深度, 函数 `startElement` 完成两项任务。对于文档中最顶层的标记, 我们创建一个新的 GTK Tree Item 窗口部件, 并将此窗口部件的指针存放在全局变量 `item` 中(在下面的程序之后讨论这一代码段):

```
if (*depthPtr == 1) {
    item = gtk_tree_item_new_with_label((char *)name);
    gtk_signal_connect(GTK_OBJECT(item), "select",
```

```

        GTK_SIGNAL_FUNC(item_signal_callback),
        "top-level item select");
    gtk_signal_connect(GTK_OBJECT(item), "deselect",
        GTK_SIGNAL_FUNC(item_signal_callback),
        "top-level item deselect");
    // a tree item can hold any number of items; add new item here:
    gtk_tree_append(GTK_TREE(tree), item);
    gtk_widget_show(item);
    // Create this item's subtree:
    subtree[*depthPtr] = gtk_tree_new();
    gtk_tree_set_view_mode(GTK_TREE(subtree[*depthPtr]),
        GTK_TREE_VIEW_ITEM);
    gtk_tree_item_set_subtree(GTK_TREE_ITEM(item),
        subtree[*depthPtr]);
}

```

在处理最顶层的 XML 标记时，我们使用全局变量 `tree` 访问 GTK Tree 窗口部件，通过这一途径将新创建的树项目添加到 GTK 树中。和函数 `handleElementData` 一样，为“select”和“deselect”操作设置信号处理函数回调函数，创建新的 GTK Tree 窗口部件并存放在数组 `subtree` 中。需要注意的是，数组 `subtree` 在此程序中作为栈数据结构使用，而分析深度相当于栈指针。这个新树设置为新创建的 GTK Tree Item 窗口部件的子树。

如果当前的分析深度大于 1（例如，处理的不是文档中最顶层的标记时），那么处理过程比处理 XML 文档中的最顶层标记要简单（在下面的程序之后我们讨论这一代码段）：

```

if (*depthPtr > 1) {
    GtkWidget * subitem = gtk_tree_item_new_with_label
        ((char *)name);
    subtree[*depthPtr] = gtk_tree_new();
    gtk_signal_connect(GTK_OBJECT(subitem), "select",
        GTK_SIGNAL_FUNC(item_signal_callback),
        "tree item select");
    gtk_signal_connect(GTK_OBJECT(subitem), "deselect",
        GTK_SIGNAL_FUNC(item_signal_callback),
        "tree item deselect");
    // Add this sub-tree it to its parent tree:
    gtk_tree_append(GTK_TREE(subtree[*depthPtr - 1]), subitem);
    gtk_widget_show(subitem);
    gtk_tree_item_set_subtree(GTK_TREE_ITEM(subitem),
        subtree[*depthPtr]);
}

```

在处理结束标记（例如“</author>”）时，分析器 `expat` 会调用函数 `endElement`。函数 `endElement` 代码很简单，将分析深度计数器减 1 即可，如下所示：

```

void endElement (void *userData, const char *name) {
    int *depthPtr = userData;
    *depthPtr -= 1;    // decrement stack pointer
}

```

作为 XML 文档中分析深度的例子,考虑下列 XML 数据(分析深度列在每行的行尾):

<book>	1
<title>Kito bit the cat</title>	2
<author>	2
<name>	3
<last_name>Smith</last_name>	4
<first_name>Joshua</first_name>	4
</name>	3
</author>	2
</book>	1

main 函数完成两项任务:启动 XML 分析器;对 GTK 窗口部件初始化。我们已经介绍过 XML 分析器的实现中处理元素打开标记、元素正文数据和元素关闭标记的回调函数。这些回调(或称信号处理函数)函数创建 GTK 树元素窗口部件,但创建最顶层的 GTK Tree 窗口部件和顶层窗口的任务由 main 函数负责。后面的讨论使用的代码段来自 main 函数的实现。

main 函数使用 James Clark 的 expat 库创建一个新的 XML 分析器对象,如下所示:

```
XML_Parser parser = XML_ParserCreate (NULL);
```

XMLviewer 应用程序使用带滚动视图区域的顶层窗口。下面的代码段指明如何设置带滚动栏的滚动区域,这一区域可在必要时自动激活。我们定义了两个指向 GtkWidget 的指针变量:window 和 scrolled_win,并调用标准的 GTK 函数 gtk_init,这个函数在先前的 GTK+ 示例中已有述及。Scrolled Window 窗口部件用 gtk_scrolled_window 函数创建,此对象将添加到变量 window 所引用的 GtkWidget 对象中。函数 gtk_scrolled_window_set_policy 可以用于设置滚动栏的自动处理(在这里我们就是这样做的),也可以将滚动栏设置为始终保持可见。函数 gtk_widget_set_usize 用于设置 Scrolled Window 窗口部件大小的最小值。用户不能将滚动窗口的大小调整到此函数设置的最小值以下。

```
GtkWidget *window, *scrolled_win;
gtk_init(&argc, &argv);
window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
scrolled_win = gtk_scrolled_window_new(NULL, NULL);
gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(scrolled_win),
                               GTK_POLICY_AUTOMATIC,
                               GTK_POLICY_AUTOMATIC);
gtk_widget_set_usize(scrolled_win, 150, 200);
gtk_container_add(GTK_CONTAINER(window), scrolled_win);
gtk_widget_show(scrolled_win);
```

GTK 库包含一个名为 gtk_main_quit 的功能回调(或信号处理)函数,此函数可用于彻底关闭任何 GTK+应用程序。下面的代码行将此函数设置为顶层窗口的窗口删除事件的信号处理函数。若没有下列代码行,当用户单击窗口的关闭按钮时应用程序不会彻底终止。

```
gtk_signal_connect(GTK_OBJECT(window), "delete_event",
                  GTK_SIGNAL_FUNC(gtk_main_quit), NULL);
```

要为添加到 GTK 容器的窗口部件之间留下空间，可以使用函数 `gtk_container_border_width` 设置窗口部件间的像素数目。通过下列代码，`main` 函数在 Tree 窗口部件周围设置了宽度为 5 个像素的边界：

```
gtk_container_border_width(GTK_CONTAINER(window), 5);
```

下列代码使用函数 `gtk_tree_new` 创建 GTK Tree 窗口部件，将新创建的 Tree 窗口部件添加到滚动窗口中：

```
tree = gtk_tree_new();
gtk_scrolled_window_add_with_viewport
(GTK_SCROLLED_WINDOW(scrolled_win), tree);
```

GTK Tree 窗口部件可以设置为单个树节点选择模式或多选择模式。下列代码行将 Tree 窗口部件设置为仅允许用户每次选择一个树节点：

```
gtk_tree_set_selection_mode(GTK_TREE(tree),
                             GTK_SELECTION_SINGLE);
```

要启用多选择模式，可以将文件 `gtkenums.h` 中定义的 GTK 常量 `GTK_SELECTION_SINGLE` 替换为另一个常量 `GTK_SELECTION_MULTIPLE`。

必须为 XMLviewer 应用程序配置 XML 分析器，这个分析器用变量 `parser` 引用。下列代码行将局部变量 `depth` 设置为 `parser` 的用户数据：

```
XML_SetUserData(parser, &depth);
```

下面两行代码为 XML 配置使用已实现的三个回调函数：`startElement`、`endElement` 和 `handleElementData`：

```
XML_SetElementHandler(parser, startElement, endElement);
XML_SetCharacterDataHandler(parser, handleElementData);
```

XMLviewer 程序从 `stdin` 读取 XML 的内容（例如，若键入“XMLviewer <test.xml”则运行 XMLviewer）。下列代码段读取 XML 数据并将其传递给 `parser`：

```
do {
    size_t len = fread(buf, 1, sizeof(buf), stdin);
    done = len < sizeof(buf);
    if (!XML_Parse(parser, buf, len, done)) {
        printf("%s at line %d\n",
               XML_ErrorString(XML_GetErrorCode(parser)),
               XML_GetCurrentLineNumber(parser));
        return;
    }
} while (!done);
XML_ParserFree(parser);
```

认识这个代码段中的执行流程至关重要。当函数 `XML_Parse` 处理 XML 数据时，它会调用函数 `startElement`、`endElement` 和 `handleElementData` 处理元素开始与结束标记，以及

元素字符数据。这些函数依次完成下列功能：构造新的 GTK Tree Element 窗口部件；将这些窗口部件插入树中的正确位置。

下面两行代码显示 Top_Level 窗口，并使其能够处理事件：

```
gtk_widget_show(window);  
gtk_main();
```

27.2.4 运行 GTK+ 的 XML 显示程序

应用程序 XMLviewer 从 stdin 读取 XML 数据。要编译并执行这个示例程序，请转到 src/GTK 目录，然后键入下列命令：

```
make  
XMLviewer <test.xml
```

图 27.2 显示两个并排运行的 XMLviewer 示例程序副本。在左图中，树是折叠的；在右图中显示的则是用户对子树分支进行扩展后的 XMLviewer。

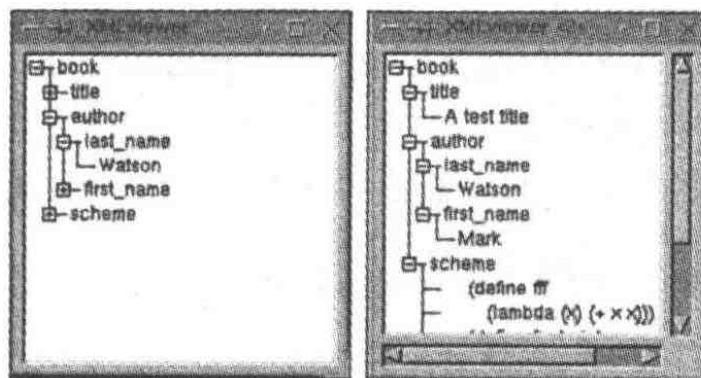


图 27.2 运行的两个 XMLviewer 示例程序副本，显示 test.xml 文件

27.3 一个使用 Notebook 窗口部件的 GUI 程序

本章的最后一个示例程序显示如何使用 GTK Notebook 窗口部件。Notebook 窗口部件是一个容器对象，包含带标签的选项卡页。本节的示例在目录 src/GTK 的文件 notebook.c 和 draw_widget.c 之中。文件 draw_widget.c 派生于 GTK+ 发布版本的 examples 目录下的 scribble-simple.c 文件。scribble-simple.c 示例程序创建一个可绘区域，并处理绘制动作的鼠标事件。在讨论完 Notebook 窗口部件示例的实现之后（使用 notebook.c 文件），我们将进一步讨论 scribble-simple 示例的实现（使用文件 draw_widget.c）。

27.3.1 Notebook 窗口部件示例程序的实现

示例程序 notebook.c 很简单，因为创建 GTK Notebook 窗口部件并添加带选项卡的页非常轻松。基本的技巧很简单：使用 gtk_notebook_new 实用工具函数创建一个 Notebook 窗口部件，然后通过下列步骤添加页：

1. 创建一个 GTK Frame 窗口部件。Frame 窗口部件是一个容器，您可以在其中添加任何内容（其他 GTK 窗口部件、定制窗口部件等等）。

2. 为选项卡创建 GTK Label 窗口部件。请注意，您可以使用任何 GTK 窗口部件作为选项卡（Label 窗口部件、Pixmap 窗口部件、Tree 窗口部件等等）。
3. 使用 GTK 实用工具函数 `gtk_notebook_append_page`，将框架（以及任何已添加到框架中的内容）和选项卡标签添加到 Notebook 窗口部件。

下面的讨论使用的代码来自文件 `notebook.c`。其中略过了前两个 GTK 示例程序中讨论过的代码（例如事件“`delete_event`”的信号处理函数，当用户单击窗口标题栏的“关闭”按钮时即发生此事件）。

和通常的做法一样，先创建一个 Top-Level Window 窗口部件，由变量 `window` 引用。下面的代码创建一个新的 Notebook 窗口部件并添加到窗口中：

```
notebook = gtk_notebook_new();
gtk_notebook_set_tab_pos(GTK_NOTEBOOK(notebook), GTK_POS_TOP);
gtk_container_add(GTK_CONTAINER(window), notebook);
gtk_widget_show(notebook);
```

这里，使用常量 `GTK_POS_TOP`（在文件 `gtkenums.h` 中定义）表示将 `notebook` 页的选项卡放在窗口部件顶部。下面是可作为函数 `gtk_notebook_set_tab_pos` 的第二个参数使用的其他可能值：

```
GTK_POS_LEFT
GTK_POS_RIGHT
GTK_POS_TOP
GTK_POS_BOTTOM
```

下面的代码摘自文件 `notebook.c`，它将 1~4 页添加到 Notebook 窗口部件中（在程序后讨论）：

```
for (i=0; i < 4; i++) {
    if (i == 0) {
        sprintf(buf1, "Draw something here with the mouse");
        sprintf(buf2, "Draw Tab");
    } else {
        sprintf(buf1, "Frame number %d:", i+1);
        sprintf(buf2, "Tab %d", i);
    }
    // Create a frame to hold anything (at all!)
    // that we might want to add to this page
    frame = gtk_frame_new(buf1);
    gtk_container_border_width(GTK_CONTAINER(frame), 10);
    gtk_widget_set_usize(frame, 240, 120);
    gtk_widget_show(frame);
    label = gtk_label_new(buf1);
    if (i == 0) {
        temp_widget = make_draw_widget(240, 120);
        gtk_container_add(GTK_CONTAINER(frame), temp_widget);
    } else {
```

```

        gtk_container_add(GTK_CONTAINER(frame), label);
        gtk_widget_show(label);
    }
    label = gtk_label_new(buf2);
    gtk_notebook_append_page(GTK_NOTEBOOK(notebook), frame, label);
}

```

在这里，添加到 Notebook 窗口部件的第一页的处理方法与另外三个不同，因为：

- 希望创建 Drawing Area 窗口部件以添加页框架。
- 对于第一页，希望在其选项卡上使用不同的标签（让用户知晓他们能在第一页绘图）。

字符数组 buf1 和 buf2 分别用于标记后面三页的内部框架（但不包括用于绘图的第一页）和页选项卡。函数 make_draw_widget 在文件 draw_widget.c 中定义，并在下一节中讨论。此函数返回一个指向 GtkWidget 的指针，此 GtkWidget 会添加到为 Notebook 窗口部件的第一页而创建的 Frame 窗口部件中。

在 4 个测试页添加到 Notebook 窗口部件中之后，下列代码依次完成下列功能：将第一页（绘制页）设置为默认的可见页、显示 Top-Level Window 窗口部件、处理事件：

```

gtk_notebook_set_page(GTK_NOTEBOOK(notebook), 0);
gtk_widget_show(window);
gtk_main();

```

27.3.2 实现 Drawing 窗口部件

文件 draw_widget.c 派生于 GTK+ 示例程序 scribble-simple.c。它创建一个一般的 GTK 窗口部件，这个窗口部件具有事件处理功能，并提供必要数据，使得用户能在窗口部件内部绘图。这个一般窗口部件并不是一种新的 GTK 窗口部件。通过调用 make_draw_widget 函数可创建 Draw 窗口部件，此函数具有下列函数原型：

```
GtkWidget * make_draw_widget(int width, int height)
```

函数 make_draw_widget 创建一个 GTK Drawing Area 窗口部件，并将信号处理函数与鼠标左按钮按下和鼠标移动事件联系起来。在下面的讨论中，我们将谈到 src/GTK 目录下的文件 draw_widget.c，但该文件是直接从 GTK 示例程序 scribble-simple.c 中派生出来的，因此这些讨论也适用于 scribble-simple.c。信号处理函数 configure_event 用于创建 off-screen 绘制的 pixmap。对于“无抖动”的动画，通常最佳的方式是在 off-screen 的缓冲区中（或 pixmap）先完成绘制操作，然后将像素图一步复制到窗口中。这种技术常常应用于视频游戏、字处理软件、绘图工具等。下列代码行创建一个新的像素图：

```

pixmap = gdk_pixmap_new(widget->window,
                        widget->allocation.width,
                        widget->allocation.height,
                        -1);

```

Drawing Area 窗口部件的地址将传递给函数 `configure_event`；这个指针指向一个 `GtkWidget` 对象，用于获得 `pixmap` 的宽度和高度等必要信息。需要注意的是，如果 `pixmap` 已经存在，那么在创建新的 `pixmap` 之前，函数 `configure_event` 首先要释放先前的 `pixmap`。在调整 `pixmap` 大小时也会调用函数 `configure_event`。在 Drawing Area 窗口部件的一部分或全部显示时则会调用信号函数 `expose_event`，此函数将 `pixmap` 复制到窗口部件的可见部分。

Drawing Area 窗口部件中的一个 X-Y 坐标位置会传递给函数 `draw_brush`。此函数在 `pixmap` 中绘制一小块黑色矩形，然后将 `pixmap` 复制到 Drawing Area 窗口部件中，如下所示：

```
Void draw_brush (GtkWidget *widget, gdouble x, gdouble y) {
    GdkRectangle update_rect;

    update_rect.x = x - 2; // 2 was 5 in original GTK example
    update_rect.y = y - 2; // 2 was 5 in original GTK example
    update_rect.width = 5; // 5 was 10 in original GTK example
    update_rect.height = 5; // 5 was 10 in original GTK example
    gdk_draw_rectangle (pixmap,
                        widget->style->black_gc,
                        TRUE,
                        update_rect.x, update_rect.y,
                        update_rect.width, update_rect.height);
    gtk_widget_draw (widget, &update_rect);
}
```

函数 `gtk_widget_draw` 在指定的窗口部件中触发一个 `Expose` 事件（通过第一个参数指定）。在这个程序中，此 `Expose` 事件会触发文件 `draw_widget.c`（以及文件 `scribble_simple.c`）中的 `expose_event` 函数调用。当鼠标按键被按下时，系统会调用信号处理函数 `button_press_event`，但当按键数目为 1 且 `pixmap` 已经在函数 `configure_event` 中设置的时候，此函数仅在 Drawing Area 窗口部件中进行绘制：

```
gint button_press_event (GtkWidget *widget, GdkEventButton *event)
{
    if (event->button == 1 && pixmap != NULL)
        draw_brush (widget, event->x, event->y);
    return TRUE;
}
```

信号处理函数 `motion_notify_event` 与 `button_press_event` 类似，但处理的是鼠标移动事件（以缩略形式列出）：

```
gint motion_notify_event (GtkWidget *widget, GdkEventMotion *event)
{
    if (event->state & GDK_BUTTON1_MASK && pixmap != NULL)
        draw_brush (widget, event->x, event->y);
    return TRUE;
}
```


结构 `GdkEventButton` 和 `GdkEventMotion` 在文件 `gdktypes.h` 中定义。缩略形式如下所示:

```
struct _GdkEventMotion { // partial definition:
    GdkEventType type;
    GdkWindow *window;
    gint8 send_event;
    guint32 time;
    gdouble x;
    gdouble y;
    gdouble pressure;
    guint state;
};

struct _GdkEventButton { // partial definition:
    GdkEventType type;
    GdkWindow *window;
    gint8 send_event;
    guint32 time;
    gdouble x;
    gdouble y;
    guint state;
    guint button;
};
```

函数 `make_draw_widget` 比较简单, 但它为 `expose`、`configure`、鼠标 `motion` 和鼠标 `button press` 事件提供了信号处理函数 (或称回调函数) 设置的示例:

```
GtkWidget * make_draw_widget(int width, int height) {
    GtkWidget *drawing_area;
    drawing_area = gtk_drawing_area_new ();
    gtk_drawing_area_size (GTK_DRAWING_AREA (drawing_area), width,
height);
    gtk_widget_show (drawing_area);
    /* Signals used to handle backing pixmap */
    gtk_signal_connect (GTK_OBJECT (drawing_area), "expose_event",
                        (GtkSignalFunc) expose_event, NULL);
    gtk_signal_connect (GTK_OBJECT (drawing_area),
                        "configure_event",
                        (GtkSignalFunc) configure_event, NULL);
    /* Event signals */
    gtk_signal_connect (GTK_OBJECT (drawing_area),
                        "motion_notify_event",
                        (GtkSignalFunc) motion_notify_event, NULL);
    gtk_signal_connect (GTK_OBJECT (drawing_area),
                        "button_press_event",
                        (GtkSignalFunc) button_press_event, NULL);
    gtk_widget_set_events (drawing_area, GDK_EXPOSURE_MASK
```

```

        | GDK_LEAVE_NOTIFY_MASK
        | GDK_BUTTON_PRESS_MASK
        | GDK_POINTER_MOTION_MASK
        | GDK_POINTER_MOTION_HINT_MASK);

    return drawing_area;
}

```

函数 `gtk_drawing_area_size` 设置 Drawing Area 窗口部件的大小。函数 `gtk_widget_set_events` 将绘图区设置为从 GTK 实用工具函数 `gtk_main` 中的 GTK 事件处理代码中接收事件信号。

27.3.3 运行 GTK Notebook 窗口部件的示例程序

将目录转到 `src/GTK`, 然后键入下列命令即可编译并运行 Notebook 窗口部件示例程序:

```

make
notebook

```

图 27.3 显示并排运行的两个 Notebook 窗口部件示例程序。在图的左侧, 包含 Drawing Area 窗口部件的第一页被选中; Drawing Area 窗口部件中记录了作者姓名的首字母缩写。图右侧显示的是 Notebook 的第二页被选中的情形。

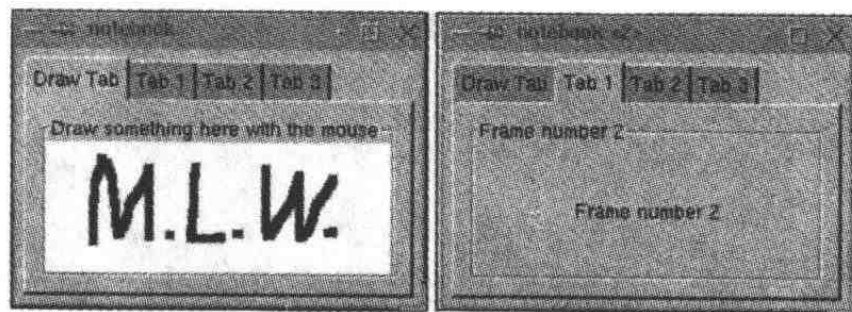


图 27.3 并排运行的两个 Notebook 窗口部件示例程序, 显示 notebook 中两个不同的页

27.4 通过其他编程语言使用 GTK+

GTK+ 还能和其他许多种编程语言绑定。其中某些编程语言的绑定只提供了对 GTK+ 一个子集的访问。那些没有提供全部特性的绑定一般都处于开发状态之中, 在将来会提供全部功能。但是, 大多数 GTK+ 的绑定都是功能完整的。特别值得一提的有 GTK+ 用于 C++、Perl 和 Python 的绑定, 本节我们对它们做快速的介绍。

为了展示怎样以不同的语言使用 GTK+, 我用每种语言实现了同一个示例程序。这个程序是经典的“Hello World”程序的 GTK+ 版本。它将用一个按钮打开一个窗口, 出现 Hello World 字样。图 27.4 显示出了 Python 版本的 Hello World 程序。单击这个按钮会在控制台打印 Hello World, 随后结束程序。当然, 这是一个很小的程序, 惟一的目的是用来说明多种绑定是怎样工作的。

本节讨论的所有软件包都可以从 GTK+ 小组的 Web 站点 <http://www.gtk.org/> 下载得到。某些软件包还有它们自己的 Web 页面, 这都可以从 GTK+ 的 Web 站点链接过去。几乎所有

的软件包都包含了出色的文档和示例程序。通过使用这些资源，你应该能够立即将 GTK+ 融入自己的程序之中。

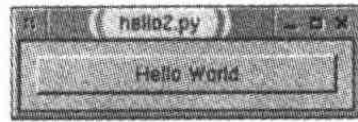


图 27.4 一种 GTK+ 版本的 Hello World

27.4.1 通过 C++ 使用 GTK+

C++ 可以使用几种 GTK+ 绑定的软件包。最完整并且功能最全的是 GTK--。它提供了一种出色的面向对象接口，可以访问 GTK+ 软件包所有的特性。GTK-- 提供了超过 180 种以上的类，可以用来创建具有 GTK+ 功能的应用程序。参考程序清单 27.3，它提供了 C++ 版本的 Hello World 程序。

程序清单 27.3 使用 C++ 的 Hello World 程序的 GTK+ 实现

```
#include <iostream>
#include <gtk--/button.h>
#include <gtk--/main.h>
#include <gtk--/window.h>

using SigC::slot;

void destroy_handler() {
    Gtk::Main::quit();
}

class HelloWorld : public Gtk::Window
{
    Gtk::Button button;

    void hello() {
        cout << "Hello World" << endl;
    }

    virtual int delete_event_impl(GdkEventAny *event) {
        return true;
    }

public:
    HelloWorld() : Gtk::Window(GTK_WINDOW_TOPLEVEL),
                  button("Hello World") {
        destroy.connect(slot(&destroy_handler));

        set_border_width(10);

        button.clicked.connect(slot(this, &HelloWorld::hello));
        button.clicked.connect(destroy.slot());
        add(button);
    }
};
```

```

        show_all();
    }
};

int main (int argc, char *argv[])
{
    Gtk::Main kit(argc, argv);

    HelloWorld helloworld;

    kit.run();
    return 0;
}

```

27.4.2 通过 Perl 使用 GTK+

GTK-Perl 软件包提供了对 GTK+和 Gnome 的 GTK+绑定。这个软件包使用 Perl 的面向对象功能提供它的 GTK+接口。Perl 程序员应该对这种绑定感到很熟悉，而且能够立即使用 GTK+开始工作。

在这个软件包中包含了许多良好的例程。它们可以作为你自己程序的基础。参考程序清单 27.4，它提供了 Hello World 程序的 Perl 实现。

程序清单 27.4 使用 Perl 的 Hello World 程序的 GTK+实现

```

use Gtk;

use vars qw($window $button);

sub hello{
    Gtk->print("hello world\n");
    destroy $button;
    destroy $window;
    exit;
}

init Gtk;

$window = new Gtk::Widget "GtkWindow",
    GtkWindow::type          => -toplevel,
    GtkWindow::title         => "hello world",
    GtkWindow::allow_grow    => 1,
    GtkWindow::allow_shrink  => 1,
    GtkContainer::border_width => 10;

$button = new_child $window "GtkButton",
    GtkButton::label          => "hello world",
    GtkObject::signal::clicked => "hello",
    GtkWidget::visible       => 1;

show $window;

main Gtk;

```

27.4.3 通过 Python 使用 GTK+

通过 Python 使用 GTK+ 非常容易，因为 PyGTK 软件包提供了出色的绑定功能。这个软件包在你的程序中提供了两种访问 GTK+ 的方法：_gtkmodule 和 gtk.py。但是，_gtkmodule 得到了人们的反对，所以你应该使用后者来编写新程序。程序清单 27.5 给出了使用 gtk.py 的 Hello World 例程。

程序清单 27.5 使用 Python 的 Hello World 程序的 GTK+ 实现

```
#!/usr/bin/env python

# this is a translation of "Hello World III" from the GTK manual,
# using gtk.py

from gtk import *

def hello(*args):
    print "Hello World"
    window.destroy()

def destroy(*args):
    window.hide()
    mainquit()

window = GtkWindow(WINDOW_TOPLEVEL)
window.connect("destroy", destroy)
window.set_border_width(10)

button = GtkButton("Hello World")
button.connect("clicked", hello)
window.add(button)
button.show()

window.show()

mainloop()
```

27.5 GTK+ 的 RAD 工具

GTK+ 最好的应用编程界面是 Glade。Glade 是免费的，而且在 GPL 下发布，它包含了对 GTK+ 和 Gnome 的支持。你可以在 Glade 的 Web 站点 <http://glade.pn.org> 找到 Glade 软件包以及相关信息。

Glade 具有许多功能和特性。它能产生 C、C++、Ada95、Python 和 Perl 的源代码。它提供了一种很直观的界面能够让你快速而方便地构造非常复杂的用户界面。参见图 27.5，它显示了 Glade 会话的外观。

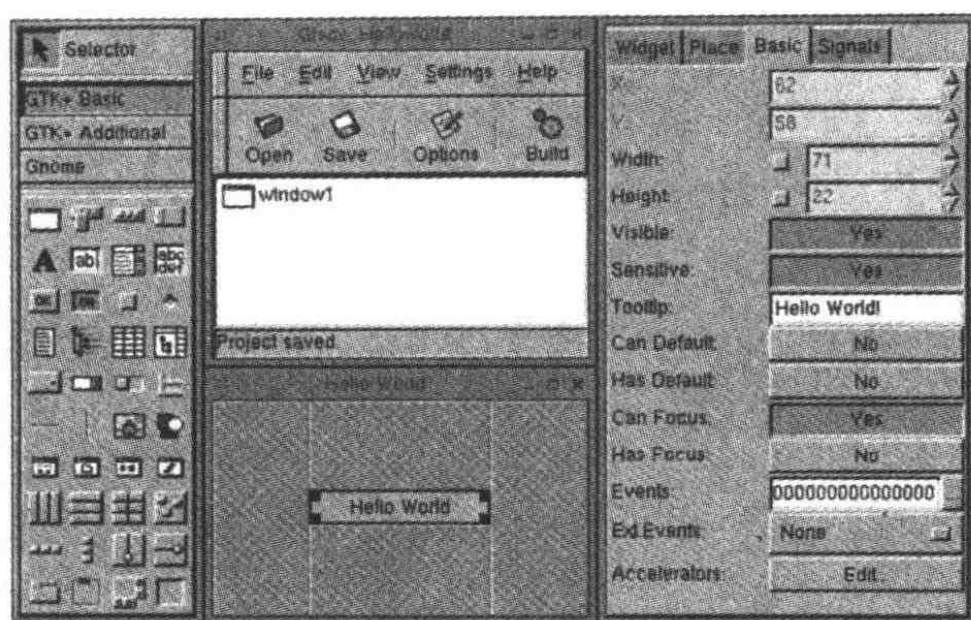


图 27.5 使用 Glade 构造 GTK+的图形用户界面

27.6 小 结

这样简短的一章内容并不意味着涵盖了 GTK+的所有内容；相反，本章只是向你介绍了 GTK+编程的基础知识，并且通过 XMLviewer.c 和 notebook.c 程序提供了趣味性的示例。我要（再次）建议你访问 GTK+官方的 Web 站点 <http://www.gtk.org/>，并且阅读 GTK+的教程。你也可能希望安装最新版的 GTK+软件包并且获得示例程序。

GTK+是一种非常强大的工具包，它提供了丰富的窗口部件，并为基于 X Windows 的 GUI 编程提供了一种高层次的方法。所有最流行的编程语言都能和它绑定。这些特色，加上诸如 Glade 这样的 RAD 工具让 GTK+成为一种非常引人注目的开发工具。最重要的是，它是在 LGPL 下发布的，因此无论在自由软件中使用还是用于开发商业软件，它都是免费的。

对 GTK+的支持和开发不大可能会停止或减弱，因为它得到了几家主要的 Linux 发布商，比如 Red Hat 的支持。GTK+也是 Gnome 桌面的关键部件，Gnome 得到了 Gnome 基金会的支持，基金会的成员包括 Red Hat 以及其他巨头比如 Sun。这必然会让 GTK+在开发人员和用户中间更加流行。

第 28 章 使用 Qt 进行 GUI 编程

用于图形用户界面编程的 Qt C++ 类库是由 Troll Tech——一家挪威公司设计和编写的。Qt 是一种跨平台的库，它支持 X Windows 和 Microsoft Windows。Qt 还有一种用于嵌入式系统的“简版”。

经过几年的演变发展，现在 Qt 成为一种非常成熟的 C++ 类库，除了支持 GUI 元素和窗口部件以外，它还能支持许多其他东西。它还支持 Unicode、XML、套接口、线程、OpenGL 等等。特别是在和桌面环境 KDE 结合在一起之后，Qt 既是一种非常良好的开发环境，也是一种出色的目标平台。如果你是一位 C++ 程序员，你可能会发现 Qt 能够满足你对一种非常高层次的 GUI 库的需求。

Qt 的 2.2 自由版本及其后续版本都在两种许可证协议下发布。这两种许可证为 QPL 和 GPL。QPL 或多或少和 GPL 兼容；参考第 34 章了解对许可证的深入讨论。因为 Qt 的 2.2 自由版本是在 GPL 许可证下发布的，所以它可以免费用于编写非商业性的应用。正如名字中“自由版本”所体现的那样，同时还存在着一个商业版本的库。如果你要开发商业的、源代码不公开的应用，则可以从 Troll Tech 购买一个许可证。参考他们的 Web 站点 <http://www.trolltech.com> 了解更多信息。

Qt C++ 类库规模很大，在本章这样短的篇幅中不可能完整地介绍它。详细的文档可以从 <http://doc.trolltech.com> 获得。在那里，你还可以找到几个趣味性的例程作为你自己的程序的基础来使用。

我希望本章中的简短示例能够鼓励你进一步深入研究标准 Qt 发布版本提供的例程。另外，你应该学会如何使用 Qt C++ 类库提供的许多随手可用的用户界面部件。在本章的末尾，你会看到为了满足应用特定的需要而使用新的 C++ 类把 Qt 窗口部件组合起来有多么容易。

在第 27 章已经看到了如何使用 GTK 用 C 语言编写 X Windows 应用程序。而在本章，你将看到，对于 C++ 程序员来说，使用 Qt 为 Linux 编写 X Windows 应用程序可能更简单。如果你愿意用 C 而不是 C++ 编程，那么可能会跳过本章而使用 Athena 窗口部件、Motif 窗口部件或 GTK+ 编程。

在本章中，将了解如何完成下面的工作：

- 通过重载 Qt 窗口部件基类 `QWidget` 中的事件方法来处理事件（例如事件 `mousePressEvent`）。
- 通过使用 Qt `signal` 和 `slots` 来处理事件。Qt 发布版本中提供了一个 C++ 预处理器 `moc`，它可以自动生成用于信号和槽的 C++ 代码。
- 通过组合现有的窗口部件类开发新的 Qt 窗口部件。

本章使用了三个简短的例子。第一个程序演示如何在窗口中画图，如何通过重载 `QWidget` 事件方法来处理事件。`QWidget` 类是所有其他 Qt 窗口部件类的 C++ 基类。这个例子源自于 Qt 的例程 `Connect`，它可以在 Qt 发布版本的 `examples` 目录下找到。第二个例子

使用的 Qt 窗口部件类 (QLCDNumber) 同样也可以在 Qt 的网上教程示例中找到。

对于本章的第二个例程, 我们派生出一种新的 C++ 窗口部件类 StateLCDWidget, 加入两个新的槽方法, 分别用来增加和减少窗口部件中显示的数值。其他窗口部件能够向两个槽中的任何一个发送信号, 以改变显示值。我们将看到一个示例, 在示例中来自两个 Push Button 窗口部件的信号(事件)将绑定到 StateLCDWidget 类中的两个槽上。本章最后一个示例将重新实现 GTK+ 那章中的程序 XMLviewer。

本章里的示例是从 Troll Tech Qt 的示例程序和教程部分衍生而来, 所以下面的(代表性的)版权声明也同样适用于本章的例程:

```

/*****
** $Id:qt/examples/drawlines/connect.cpp 2.2.0 edited 2000-08-31 $
**
** Copyright (C) 1992-2000 Trolltech AS. All rights reserved.
**
** This file is part of an example program for Qt. This example
** program may be used, distributed and modified without limitation.
**
*****/

```

28.1 通过重载 QWidget 类方法处理事件

这个示例程序很短(大约 50 行代码)但它演示了如何创建一个简单的、使用 Qt 窗口部件的主应用程序窗口(文件 man.cxx), 以及如何从 Qt QWidget 类(文件 draw.hxx 和 draw.cxx)派生一个新的窗口部件类。在编写示例程序之前, 首先介绍一下 C++ QWidget 类中最常用的公用接口。在本章的稍后部分我们会看到 Qt 中使用信号与槽的事件处理方案。两种事件处理方案可以在同一个程序中混用。

28.1.1 QWidget 类概述

QWidget 类是所有其他 Qt 窗口部件的 C++ 基类, 它提供一组公用的 API 来控制窗口部件并设置窗口部件参数。QWidget 类定义多个事件处理方法, 这些方法可在派生类(在 Qt 发布版本的 qwidget.h 文件中)中重载, 如下所示:

```

virtual void mousePressEvent( QMouseEvent *);
virtual void mouseReleaseEvent( QMouseEvent *);
virtual void mouseDoubleClickEvent( QMouseEvent *);
virtual void mouseMoveEvent( QMouseEvent *);
virtual void wheelEvent( QWheelEvent *);
virtual void keyPressEvent( QKeyEvent *);
virtual void keyReleaseEvent( QKeyEvent *);
virtual void focusInEvent( QFocusEvent *);
virtual void focusOutEvent( QFocusEvent *);
virtual void enterEvent( QEvent *);

```



```

virtual void leaveEvent( QEvent *);
virtual void paintEvent( QPaintEvent *);
virtual void moveEvent( QMoveEvent *);
virtual void resizeEvent( QResizeEvent *);
virtual void closeEvent( QCloseEvent *);
virtual void dragEnterEvent( QDragEnterEvent *);
virtual void dragMoveEvent( QDragMoveEvent *);
virtual void dragLeaveEvent( QDragLeaveEvent *);
virtual void dropEvent( QDropEvent *);
virtual void showEvent( QShowEvent *);
virtual void hideEvent( QHideEvent *);
virtual void customEvent( QCustomEvent *);

```

从名称上看，这些方法的用途一目了然，但事件参数类型需要简单解释一下。
QMouseEvent 类具有下列（一部分）公用接口：

```

class QMouseEvent : public QEvent { // mouse event
public:
    int x();           // x position int widget
    int y();           // y position int widget
    int globalX();     // x relative to X server
    int globalY();     // y relative to X server
    int button();      // button index (starts at zero)
    int state();       // state flags for mouse
    // constructors, and protected/private interface is not shown
};

```

本节的例子中仅用到了鼠标事件。我们将捕获鼠标的按下和移动事件发生时的 *x,y* 坐标。在本章里我们没有用到 **QKeyEvent**, **QEvent**, **QFocusEvent**, **QPaintEvent**, **QMoveEvent**, **QResizeEvent** 或 **QCloseEvent** 类，但你可以在 Qt 发布版本的 *src/kernel* 子目录下的 *qevent.h* 文件中查看这些类的头文件内容。由于篇幅所限，本章无法涵盖 Qt 中使用的所有事件类型，但你可以参考 *qevent.h* 文件快速找到这些类的头信息。

QSize 类具有下列（部分的）公用接口：

```

class QSize {
public:
    QSize() { wd = ht = -1; }
    QSize( int w, int h );
    int width() const;
    int height() const;
    void setWidth( int w );
    void setHeight( int h );
    // Most of the class definition not shown.
    // See the file src/kernel/qsize.h in the Qt distribution
};

```

QSize 的类接口在 Qt 发布版本的 *src/kernel* 目录下的文件 *qsize.h* 中定义。**QWidget** 类定义了多个实用工具方法，可用于控制或查询窗口部件的状态（摘自 Qt 发布版本的文件

QWidget.h), 如下所示:

```
int x();
int y();
QSize size();
int width();
int height();
QSize minimumSize();
QSize maximumSize();
void setMinimumSize(const QSize &);
void setMinimumSize(int minw, int minh);
void setMaximumSize(const QSize &);
void setMaximumSize(int maxw, int maxh);
void setMinimumWidth(int minw);
void setMinimumHeight(int minh);
void setMaximumWidth(int maxw);
void setMaximumHeight(int maxh);
```

这些公用方法可用于:

- 获得某个容器内的 Qt 窗口部件的 x,y 坐标位置
- 获得宽度和高度
- 获得窗口部件大小的最小和最大值 (可以指定最小和最大值的大小及形状)

28.1.2 实现 DrawWidget 类

通过两个步骤实现示例程序。在本节要编写 DrawWidget 的 C++ 类。在下一节再编写使用 DrawWidget 的主程序。在本书的 Web 站点 src/Qt/events 目录下的 draw.hxx 文件包含了 DrawWidget 的 C++ 类接口。这个文件从 Qt 发布版本 examples 目录下的文件 connect.h 派生。类定义需要 QWidget, QPainter 和 QApplication 类的定义:

```
#include <QWidget.h>
#include <QPainter.h>
#include <QApplication.h>
```

QPainter 类封装了绘图属性, 如笔和刷的样式、背景色、前景色和剪贴区域等。QPainter 类提供基本的绘图操作, 如画点、线以及几何图形。QPainter 类还提供高级的绘图操作, 如贝赛尔 (Bezier) 曲线、文本和图像等。你可以在 Qt 发布版本的 src/kernel 目录下的文件 QPainter.h 之中找到 QPainter 的 C++ 类接口。

QApplication 类对数据进行封装, 并提供 X Windows 顶层应用程序的操作。这个类跟踪添加到某个应用程序的所有 Top_Level 的窗口部件。所有的 X Windows 事件均由 QApplication 类的 exec 方法处理。

示例 DrawWidget 类以公开方式从 QWidget 派生, 它重载了 3 种保护方法: paintEvent, mousePressEvent 和 mouseMoveEvent:

```
class DrawWidget : public QWidget {
public
```

```

    DrawWidget(QWidget *parent=0, const char *name=0);
    ~DrawWidget();
protected:
    void paintEvent(QPaintEvent *);
    void mousePressEvent(QMouseEvent *);
    void mouseMoveEvent(QMouseEvent *);
private:
    QPoint *points;           // point array
    QColor *color;            // color value
    int count;                // count = number of points
};

```

数组 `points` 用于存储窗口部件中鼠标事件的 `x,y` 坐标。类变量 `color` 用于定义定制的绘制颜色。变量 `count` 用于对收集的指针进行计数。

文件 `draw.cxx` 包含 `DrawWidget` 类的实现。include 文件 `draw.hxx` 包括下列类头信息说明：

```
#include "draw.hxx"
```

使用容量为 3000 的指针数组记录窗口部件内的鼠标位置。数组在类的构造函数中初始化，已存储指针的计数设置为零，如下所示：

```

const int MAXPOINTS = 3000; // maximum number of points
DrawWidget::DrawWidget(QWidget *parent, const char *name)
    : QWidget(parent, name) {
    setBackgroundColor(white);           // white background
    count = 0;
    points = new QPoint[MAXPOINTS];
    color = new QColor(250, 10, 30); // Red, Green, Blue
}

```

类的析构函数简单地释放指针数组：

```

DrawWidget::~~DrawWidget() {
    delete[] points; // free storage for the collected points
}

```

在类示例 `DrawWidget` 中定义的方法 `paintEvent` 重载了基类 `QWidget` 中的定义。类 `QPainter` 的新实例用于定义图形环境，以及提供方法 `drawRect`，此方法绘制在 `DrawWidget` 中以鼠标位置为中心的小矩形。数组 `points` 在鼠标按下和鼠标移动事件的方法中填充。

```

void DrawWidget::paintEvent(QPaintEvent *) {
    QPainter paint(this);
    paint.drawText(10, 20, "Click the mouse buttons,
        or press button and drag");
    paint.setPen(*color);
    for (int i=0; i<count; i++) { // connect all points
        paint.drawRect(points[i].x()-3, points[i].y()-3, 6, 6);
    }
}

```

类 `DrawWidget` 中定义的方法 `mousePressEvent` 重载了基类 `QWidget` 中的定义。此方法有两个功能：在数组 `points` 中记录当前鼠标位置并立即以此位置为中心绘制一个红色的（在变量 `color` 中定义）小矩形。

```
void DrawWidget::mousePressEvent(QMouseEvent * e) {
    if(count < MAXPOINTS) {
        QPainter paint(this);
        points[count] = e->pos();          // add point
        paint.setPen(*color);
        paint.drawRect(points[count].x()-3,points[count].y()-3,6,6);
        count++;
    }
}
```

类 `DrawWidget` 中定义的方法 `mouseMoveEvent` 重载了基类 `QWidget` 中的定义。与鼠标按下方法类似，此方法有两个功能：在数组 `points` 中记录当前鼠标位置并立即以此位置为中心绘制一个红色的（在变量 `color` 中定义）小矩形。

```
void DrawWidget::mouseMoveEvent(QMouseEvent *e) {
    if(count < MAXPOINTS) {
        QPainter paint(this);
        points[count] = e->pos();          // add point
        paint.setPen(*color);
        paint.drawRect(points[count].x()-3,points[count].y()-3,6,6);
        count++;
    }
}
```

28.1.3 测试 `DrawWidget`

示例的绘图窗口部件实现很简单。主程序完成下列功能：创建应用程序窗口部件、添加绘图窗口部件、处理 X Windows 事件（其实现更简单）等。包含主测试函数的测试文件名为 `main.cxx`，放在目录 `src/Qt/events` 目录下。这里只需要一个 `include` 文件，即 `draw.hxx` 头文件，因为 `draw.hxx` 中包含了必要的 Qt 头文件：

```
#include "draw.hxx"
```

函数 `main` 分别定义了类 `QApplication` 和示例类 `DrawWidget` 的一个实例。`QApplication` 的方法 `setMainWidget` 将绘图窗口部件指定为应用程序的 `Top_Level` 窗口部件。方法 `show` 从类 `QWidget` 继承，功能是使绘图窗口部件可见。`QApplication` 的方法 `exec` 处理 X Windows 事件：

```
int main(int argc, char **argv) {
    QApplication app(argc, argv);
    DrawWidget draw;
    app.setMainWidget(&draw);
    draw.show();
    return app.exec();
}
```

从这个简短的示例程序中可以看出，Qt 类库确实封装了 X Windows 编程的很多复杂性。如果有必要，你可以将低层的 Xlib 编程与 Qt 类库的使用相混合，但很少需要这么做，因为 Qt 库支持绘图原语。

图 28.1 显示了支持随手画的绘图程序。

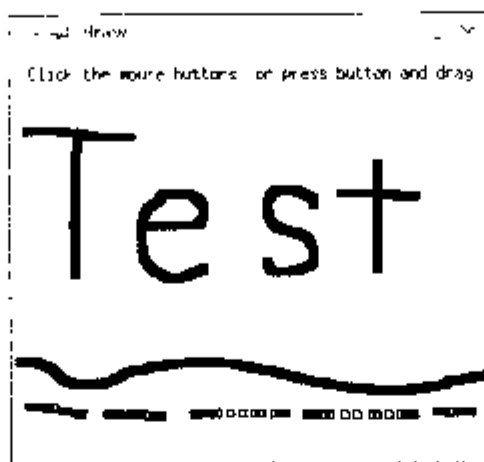


图 28.1 main.cxx 程序使用了 DrawWidget

28.2 使用 Qt 槽和信号处理事件

对 Qt 槽和信号的使用是一种高层次接口，用于协调不同 Qt 窗口部件中的事件和操作。使用 Qt 槽和事件有些复杂，因为需要使用一个 C++ 预处理程序——Qt 工具 moc (Meta Object Compiler) ——为这种高层次的事件处理自动生成附加代码。本节中的示例放在 src/Qt/signals_slots 目录下。此目录中的 Makefile 设置为：使用 moc 工具产生附加的源代码，然后编译并链接。如果你的系统配置与本书所使用的测试机不一致，Makefile 则不能正确运行。然而，这个 Makefile 是从编程教程（包含在 Qt 标准发布版本中）的 Makefile 示例中复制与编辑的。当 Qt 标准版在编译和安装时，教程里的这些 Makefile 会自动建立起来。如果这里作为示例的 Makefile 因为某个头文件或某个 X 库的位置错误而无法工作，请将它的前 20 行与 Qt 标准版中教程里的任一 Makefile 进行比较。

安装 Qt 时，无论是从你的 Linux 发布版本，还是从 www.trolltech.com 的最新版本安装，环境变量 QTDIR 均需要设置为指向安装的库、二进制工具等等。在我的系统中，QTDIR 设为 /usr/local/qt。Qt 提供 C++ 语言的一些简单扩展，它使用 \$QTDIR/bin 目录下的预处理器 moc。注意，\$QTDIR/bin 目录应该在你的路径变量 PATH 中。要使用本章的材料，必须按照 Qt 发布版本中的安装说明进行了安装。我们会在稍后介绍 moc 的用法，但在下一节我们会展示它的一个简单应用示例。

28.2.1 派生 StateLCDWidget 类

Qt C++ 类库包含许多有用的窗口部件。要看到所有不同的 Qt 窗口部件，最佳的方法是转到 Qt 发布版本的 examples 目录下，然后编译并运行所有的示例程序。当我第一次开始使用 Qt 时，这个练习大约花费了我 20 分钟的时间，它向我展示了 Qt 类库的强大功能。

本节我们将使用 Qt 窗口部件类 `QLCDNumber`。`QLCDNumber` 窗口部件在窗口部件内部用大字体显示一个数字。`QLCDNumber` 类具有用于设置所显示数字的方法，但在我们的例子中，我们希望定义两个槽，分别用于递增和递减显示的数字。稍后我们将看到，若用户在 Qt Push Button 窗口部件上单击，产生的信号很容易与这些槽连接。

要展示使用槽和信号处理事件的方法，我们需要创建一个名为 `StateLCDWidget` 新类，它包含两个槽，这些槽作为方法在类的头文件中定义：

```
void increaseValue();
void decreaseValue();
```

在下一节将会看到如何使用 Qt 窗口部件的槽；在本节，我们先实现 `StateLCDWidget` 类和这两个槽。下面的程序显示了 `StateLCDWidget` 类完整的头文件，它位于 `src/Qt/signals_slots` 目录下的 `state_lcd.hxx` 文件中（在程序后进行讨论）：

```
#include <qwidget.h>
#include <qlcdnumber.h>

class StateLCDWidget : public QWidget {
    Q_OBJECT
public:
    StateLCDWidget(QWidget *parent=0, const char *name=0);
public slots:
    void increaseValue();
    void decreaseValue();
protected:
    void resizeEvent(QResizeEvent *);
private:
    QLCDNumber *lcd;
    int value;
};
```

`include` 文件 `QWidget.h` 和 `qlcdnumber.h` 包含 Qt 类库中 C++ 类 `QWidget` 和 `QLCDNumber` 的类定义。对于实际的应用程序，类 `StateLCDWidget` 通常从类 `QLCDNumber` 派生，但对于我们这里的简单示例程序（如何定义槽），让类 `StateLCDWidget` 从更简单的 `QWidget` 类派生，并使用一种包容关系，这种方法更容易一些。类 `StateLCDWidget` 包含 `QLCDNumber` 类的一个实例。

在 `state_lcd.hxx` 文件中，我们看到下列形式上不合法的代码：

```
Q_OBJECT

public slots:
    void increaseValue();
    void decreaseValue();
```

符号 `Q_OBJECT` 使得 `moc` 工具程序生成所谓的“元对象协议”信息，通过这些信息，在运行时刻可以对对象进行检查，以确定某些类属性。对元对象协议的支持内置于某些面向对象编程系统上，如 Common LISP/CLOS。Qt 的体系结构建立在运行时刻对对象进行检

查的概念上，以支持将代码绑定到特定对象的槽——而不是一个类的所有实例。

注意： 符号 `slots` 不是 C++ 的保留字。

当 C++ 编译器编译这些代码时，符号 `slot` 在 Qt include 文件 `qwidget.h` 中未得到任何定义。使用这一窗口部件编译程序时，必须在 `Makefile` 中添加命令，以运行 Qt 工具程序 `moc`，它创建新的 C++ 源文件，此源文件带有这个类的附加代码。在了解过文件 `state_lcd.cxx` 中的 `StateLCDWidget` 类实现之后，我们再进一步了解 `moc`。要定义窗口部件 `StateLCDWidget` 和 `QLCDWidget`，我们需要两个 `include` 文件：

```
#include "state_lcd.hxx"
#include <qlcdnumber.h>
```

这个类的构造函数很简单：它调用超类 `QWidget` 的构造函数并创建 `QLCDNumber` 类的一个实例，然后指定此实例中最多显示 4 位数字：

```
StateLCDWidget::StateLCDWidget(QWidget *parent, const char *name)
    : QWidget(parent, name) {
    lcd = new QLCDNumber(4, this, "lcd");
}
```

要处理调整大小事件，方法 `resizeEvent` 是必要的，它通过调用 `QLCDNumber` 类的方法 `resize` 来完成任务：

```
void StateLCDWidget::resizeEvent(QResizeEvent *) {
    lcd->resize(width(), height() - 21);
}
```

这里，方法 `width` 和 `height` 用于计算 `QLCDNumber` 实例的大小。方法 `width` 和 `height` 引用 `StateLCDWidget`；这些方法从 `QWidget` 类中继承。类实现的其他部分包括两个方法 `increaseValue` 和 `decreaseValue` 的定义，这两个方法分别将私有变量 `value` 加 1 或减 1：

```
void StateLCDWidget::increaseValue() {
    value++;
    lcd->display(value);
}
void StateLCDWidget::decreaseValue() {
    value--;
    lcd->display(value);
}
```

现在讨论类槽的创建和 Qt 工具程序 `moc`。对于这一节定义的 `StateLCDWidget` 和下一节定义的 `UpDownWidget` 类，其中都要用到工具程序 `moc`。`Makefile` 中的下列规则指定了 `make` 程序如何处理使用信号和槽的源文件：

```
moc_state_lcd.cxx: state_lcd.hxx
    $(MOC) state_lcd.hxx -o moc_state_lcd.cxx
moc_up_down.cxx: up_down.hxx
```

```
$(MOC) up_down.hxx -o moc_up_down.cxx
```

假定\$(MOC)引用的是 Qt 发布版本的 bin 目录中，工具程序 moc 的绝对路径。如果头文件被更改，新的源文件（这里是 moc_state_lcd.hxx 或 moc_up_down.cxx）会由 moc 自动创建。文件 Makefile 中的下述规则对这些生成的源文件进行编译：

```
moc_state_lcd.o: moc_state_lcd.cxx state_lcd.hxx
moc_up_down.o: moc_up_down.cxx up_down.hxx
```

Makefile 包含从 C 或 C++ 源文件编译生成.o 文件的规则。所生成的 moc 文件包含窗口部件声明的槽及把信号绑定到其他窗口部件中包含的槽所必需的代码。

28.2.2 使用信号和槽

在上一节，我们开发了 StateLCDWidget 类，作为一个简单的类示例，它定义了可用于其他 Qt 窗口部件的槽。在这一节，我们要开发另外一个简单的窗口部件类 UpDownWidget，它包含窗口部件类 StateLCDWidget 的一个实例，以及 Qt QPushButton 窗口部件类的两个实例。一个下压式按钮将其 clicked 信号绑定到类 StateLCDWidget 实例的 decreaseValue 槽，另一个下压式按钮则将其 clicked 信号绑定到类 StateLCDWidget 实例的 increaseValue 槽。

类的头文件 up_down.hxx 需要 3 个 include 文件才能定义类 QWidget、QPushButton 和 StateLCDWidget：

```
#include <qwidget.h>
#include <qpushbutton.h>
#include "state_lcd.hxx"
```

UpDownWidget 的类定义用到了两个 C++ 编译器不支持的符号，但它们对工具程序 moc 具有特殊意义。这两个符号是 Q_OBJECT 和 signals。正如前一节中所看到的，符号 Q_OBJECT 提示工具程序 moc 生成所谓的“元对象协议信息”，以支持将代码绑定到特定对象的槽而不是一个类的所有实例。

工具程序 moc 在类定义中使用符号 signals，确定哪个方法可以通过槽连接被类中的特定实例调用。需要重点了解的是，因为某个对象已经定义了槽，应用程序可能无法将这些槽绑定到来自其他窗口部件对象的信号上。这种绑定是基于对象到对象的，而不是到类的所有实例。UpDownWidget 的类定义表明，此类中还定义了另外三个窗口部件对象：两个下压按钮对象和一个 StateLCDWidget 对象：

```
class UpDownWidget : public QWidget {
    Q_OBJECT
public:
    UpDownWidget(QWidget *parent=0, const char *name=0);
protected:
    void resizeEvent(QResizeEvent *);
private:
    QPushButton *up;
    QPushButton *down;
    StateLCDWidget *lcd;
};
```


文件 `up_down.cxx` 的类定义中包含了类构造函数和 `resizeEvent` 方法的定义。构造函数的定义指明如何将一个窗口部件的信号绑定到另一个窗口部件的槽：

```
UpDownWidget::UpDownWidget( QWidget *parent, const char *name )
    : QWidget( parent, name )
{
    lcd = new StateLCDWidget(parent, name);
    lcd->move( 0, 0 );
    up = new QPushButton("Up", this);
    down = new QPushButton("Down", this);
    connect(up, SIGNAL(clicked()), lcd, SLOT(increaseValue()) );
    connect(down, SIGNAL(clicked()), lcd, SLOT(decreaseValue()) );
}
```

方法 `connect` 从 `QObject` 继承而来，这个类是 `QWidget` 的基类。方法的函数原型如下所示，它在文件 `qobject.h` 中定义，这个文件在 Qt 发布版本的 `src/kernel` 目录下。

```
bool connect(const QObject *sender, const char *signal,
            const char *member);
```

宏 `SIGNAL` 在 `qobjectdefs.h` 中定义如下：

```
#define SIGNAL(a)      "2"#a
```

如果你在构造函数的末尾添加下列代码行：

```
printf("SIGNAL(clicked()) = |%s|\n", SIGNAL(clicked()));
```

执行构造函数时，就会看到从宏 `SIGNAL` 而来的下列输出：

```
SIGNAL(clicked()) = |2clicked()|
```

`moc` 生成的代码会识别这个字符串（由宏 `SIGNAL` 创建）并在运行时刻将它绑定到正确的信号。宏 `SLOT` 在 `qobjectdefs.h` 中定义如下：

```
#define SLOT(a) "1"#a
```

如果你在构造函数的末尾添加下列代码行：

```
printf("SLOT(increaseValue()) = |%s|\n", SLOT(increaseValue()));
```

执行构造函数时，就会看到从宏 `SIGNAL` 而来的下列输出：

```
SLOT(increaseValue()) = |1increaseValue()|
```

`moc` 生成的代码会识别这个字符串（由宏 `SLOT` 创建）并在运行时刻将它绑定到正确的成员函数。方法 `resizeEvent` 简单地调整 `StateLCDWidget` 窗口部件的大小，并重新确定两个 `Push Button` 窗口部件的位置：

```
void UpDownWidget::resizeEvent(QResizeEvent *) {
    lcd->resize(width(), height() - 59);
    up->setGeometry(0, lcd->height() + 5, width(), 22);
    down->setGeometry(0, lcd->height() + 31, width(), 22);
}
```

这里，UpDownWidget 的宽度和高度用于计算 UpDownWidget 中 3 个窗口部件的位置和大小。

28.2.3 运行信号/槽示例程序

现在已经完成了 StateLCDWidget 和 UpDownWidget 窗口部件。文件 main.cxx 包含一个简单的示例程序可测试这些窗口部件：

```
#include <qapplication.h>
#include "up_down.hxx"

int main(int argc, char **argv) {
    QApplication a(argc, argv);
    QWidget top;
    top.setGeometry(0, 0, 222, 222);
    UpDownWidget w(&top);
    w.setGeometry(0, 0, 220, 220);
    a.setMainWidget(&top);
    top.show();
    return a.exec();
}
```

这个示例程序与测试 Drawing Area 窗口部件的文件 main.cxx 略有不同，因为一个独立的 Top-Level 窗口部件要添加到 Application 窗口部件中。然后 UpDownWidget 再添加到这个 Top-Level 的窗口部件中。方法 setGeometry 在文件 qwidget.h 中定义，其原型如下：

```
virtual void setGeometry(int x, int y, int width, int height);
```

只有 Top-Level 窗口部件必须用方法 show 设置为可见。Top-Level 窗口部件包含的任何窗口部件也必须可见。图 28.2 显示了 main.cxx 中示例程序的运行情况。单击 Up 按钮可将显示值增 1；单击 Down 按钮将显示值减 1。

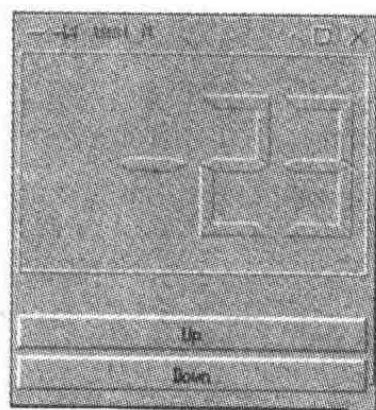


图 28.2 两个 Push Button 窗口部件的单击信号连接到 StateLCDWidget 窗口部件中的槽

28.3 用 Qt 实现 XMLview 的程序

在前面一章中介绍了 GTK+，并且实现了一个简单的 XML 查看器作为一个例程。在这一节，将实现一个类似的程序，但这一次使用 Qt 库。参考上一章有关的讨论，了解 XML 的作用及用法。

Qt 包含了几个额外的类软件包，它们提供了超出图形用户界面和窗口部件之外的功能。这些类软件包之一是用于 XML 格式的数据的处理模块。XML 模块提供了一个结构良好的 XML 分析器。Qt 中有两种使用 XML 分析器的方法。一种是 SAX2 接口，它是一种事件驱动的接口。另一种是 DOM 接口，它把 XML 文档表示成一个树型结构。将在下一节学习这两种方法。

28.3.1 SAX2: 一个用于 XML 的简单 API

Qt 的 XML 模块提供了一个 SAX2 接口。SAX2 接口常常也称为事件驱动接口。从事件驱动的意义上来讲，程序员必须提供调用自己代码的钩子函数，钩子函数在分析器处理 XML 文档时被触发，并且依赖触发器才能发生。分析器只对整个文档分析一次，并且在分析的过程中触发事件；当分析器抵达文档的结尾时就结束运行。SAX2 顺序查看文档的内容。如果你需要多次访问 XML 文档，要把所需值保存下来再重新分析整个文档。

为了使用 SAX2 接口，需要 `QXmlDefaultHandler` 类派生的子类，并提供在事件被触发时采用你自己的处理事件的方法。在基类 `QXmlDefaultHandler` 中的大多数方法默认情况下都不执行任何操作，所以你需要在自己的子类中提供这些方法的操作行为。要保证有个出错处理函数，因为它是默认情况下不执行任何操作的方法之一。

你要构造一个能够输出 XML 文件结构的类，作为一个说明如何使用 SAX2 接口的简单例程。这个例程显示出构成 XML 文件的标记并且在父标记下缩行显示子标记。这个类和下面类似：

```
Class StructureParser : public QxmlDefaultHandler
{
public:
    bool startDocument()
    {
        indent = "";
        return TRUE;
    }
    bool startElement( const QString&, const QString&, const QString&
                      qName, const QxmlAttributes& )
    {
        cout << indent << qName << endl;
        indent += " ";
        return TRUE;
    }
    bool endElement( const QString&, const QString&, const QString& )
    {
        indent.remove(0, 1);
        return TRUE;
    }
    bool error( const QxmlParseException &e )
    {
        cout << "XML Error: " << e.message() << endl;
    }
}
```

```

        return TURE;
    }
private:
    QString indent;
};

```

表 28.1 给出了类的方法及其功能的解释。

表 28.1 StructureParser 示例中使用的方法

方法	描述
startDocument	一旦分析文档的工作开始，就调用这个方法。在本例中，它只用来把 indent 变量清空成一个空字符串
startElement	分析器每遇到一个起始标记就触发这个方法。我们这个版本的程序打印标记的名称并且增加缩进的位置以防在标记中还嵌套有其他标记
endElement	分析器每次发现一个结束标记就触发这个方法。这意味着我们需要从 indent 字符串删除一个空格，因为我们已经从一级嵌套中退出来了
error	当分析文档时如果出现了一个可恢复的错误，分析器就调用这个方法。在本例中，我们将只打印出错消息然后继续分析文档

要了解有关 SAX2 接口的更多信息，参考 <http://www.megginson.com/SAX/>。

28.3.2 DOM: 文档目标对象

Qt 的 XML 模块还提供了 W3C.COM 制订的文档对象模型 (Document Object Model, DOM) Level 1 接口的一个实现，它提供了一个访问和控制 XML 文档的接口。你可以通过 DOM 接口改变 XML 文档的内容和结构。通过使用 DOM 把 XML 文档表示成一个有层次的树型结构。你可以使用处理树型数据的标准编程技术遍历它。如果你想了解有关 DOM 内部工作机制的更多信息，可以在 <http://www.w3.org/DOM/> 找到丰富的资料。

为了实现第 27 章中 XMLviewer 例程的 Qt 版本，我们将使用 DOM 和 XML 接口。为了做到这一点，将首先读取 XML 文件，再分析它，然后遍历产生的 DOM 层次结构中的节点。对于每个经过的节点，我们都把其名字或数据加入一个列表框。

这个例程实质上并不是面向对象的。故意这样做是为了保持程序尽可能的简短和清楚。如果你在现实中实现一个类似的解决方案，从代码重用的角度来看，使用面向对象技术可能是个好主意。

要做的第一项工作是创建一个 XML 对象保存我们的数据。这通过下面的代码完成：

```
QDomDocument doc("mydocument");
```

这条语句创建了一个没有内容的 XML 对象。要得到其中的内容，需要读取用户在命令行上指定的文件。这通过下面的代码完成：

```

Qfile f(argv[1]);
if (f.open(IO_ReadOnly)) {
    QTextStream t(&f);
    QString s;

```

```

while (!t.eof()) s.append(t.readLine());

doc.setContent(s);
f.close();
} else {
    cout << "Couldn't open " << argv[1] << endl;
    exit(-1);
}

```

我们使用了 Qt 内建的对文件访问的支持功能，而不是 C++ 支持的普通的流功能，因为希望本例是个 Qt 应用。接着我们尝试打开文件读取数据，如果执行成功，我们会把整个文件的内容读入到一个 QString 对象中。通过使用 setContent 方法，就能告诉 QDomDocument 对象这是我们要使用的 XML 数据。这个方法分析字符串的 XML 数据，并且构造一个 DOM 树型结构来表示我们的数据集。

然后向应用加入一个 List View 窗口部件。这个窗口部件用来以图形化方式向用户显示 XML 数据。这通过下面的代码完成：

```

QListView lv(0,argv[1]);

lv.addColumn("Data");
lv.setSorting(-1);
lv.setRootIsDecorated(TRUE);

QListViewItem lvi(&lv,"XML Data");

```

这里，创建一个新的空的 QListView 窗口部件 lv。接着用 Data 字符串加入一列。因为既想保留顺序也想保留 XML 数据的结构，所以就关闭了对窗口部件的排序处理。我们想在有子项的项的前面有个加号，就可以使用 setRootIsDecorated 方法。最后，在列表中加入一项。把它作为加入其他所有 XML 项的定位列表项。正如你将在下面的代码段中所看到的那样，这个额外的列表项简化了分析 DOM 节点的函数的编写工作。

为了在列表中填入数据项，需要遍历 DOM 对象中的 XML 数据节点。要做到这一点，这个简短的函数能够递归地经历 XML 树的每个分支。随着它经历每个节点，它也把每个节点的内容加入到列表中。这个函数的定义如下：

```

Void traversenode(QDomNode &n, QListViewItem *parent)
{
    QListViewItem *after=parent;

    while(!n.isNull()) {
        QListViewItem *parentptr= parent;

        if(!n.isText())
            parentptr=after=new
QListViewItem(parentptr,after,n.nodeName());

        if(!n.nodeValue().isNull())
            parentptr=after=new
QListViewItem(parentptr,after,n.nodeValue());
    }
}

```

```

    QDomNode child=n.firstChild();
    if(!child.isNull())
        traversenode(child,parentptr);

    n=n.nextSibling();
}
}

```

`traversenode` 函数是一个简单而直观的递归函数。它有两个参数。第一个参数是当前正在处理的 DOM 节点。第二个参数是指向列表项的指针，其中保存着这个分支的数据。这个函数将处理输入节点和它的所有兄弟节点。对于每个节点来说，函数将显示节点包含的数据的类型。如果节点不是一个文本节点，就加入一个以节点命名的列表项。如果节点包含一个值，就把这个值放入一个新列表项中。此时使用 `QListViewItem` 方法插入新列表项，作为父项的子项。如果一个节点没有任何子节点，我们就再次调用 `traversenode` 函数递归地处理它们。

第一次调用遍历函数时，需要决定应该从在 DOM 树中的哪个节点开始。我们通过下面的代码完成：

```

QDomNode child=doc.firstChild();
child=child.nextSibling();

traversenode(child,&lvi);

```

使用 `QDomDocument` 对象的 `firstChild` 方法找出我们的 DOM 层次结构中的第一个节点。对于本例来说，将跳过树型结构中最上面的节点，它们只告诉我们这是一个 XML 文档以及 XML 是哪一种版本。一旦得到了要查看的数据的顶节点，就可以调用 `traversenode` 函数。

参考程序清单 28.1 了解这个例程的全部代码。你可以用下面的命令运行这个程序：

```
./XMLviewer test.xml
```

当运行这个程序的时候，应该能看到类似图 28.3 的输出。

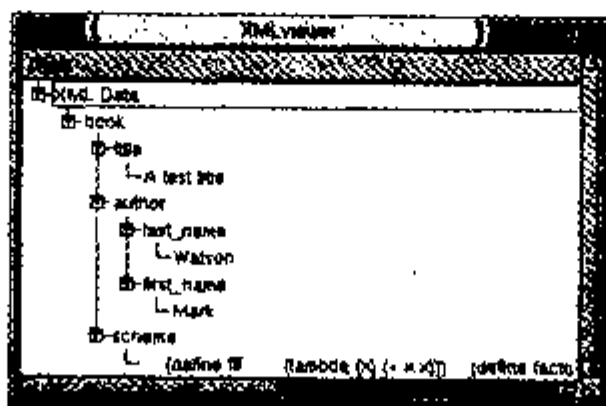


图 28.3 XMLviewer 显示了文件 test.xml 的内容

程序清单 28.1 XMLviewer 的完整 Qt 版本

```

#include <stdlib.h>
#include <iostream.h>

```

```
#include <qapplication.h>
#include <qwidget.h>
#include <qdom.h>
#include <qfile.h>
#include <qtextstream.h>
#include <qlistview.h>

void traversenode(QDomNode &n, QListViewItem *parent)
{
    QListViewItem *after=parent;

    while(!n.isNull()) {
        QListViewItem *parentptr=parent;

        if(!n.isText())
            parentptr=after=new QListViewItem(parentptr, after,
                                                n.nodeName());

        if(!n.nodeValue().isNull())
            parentptr=after=new QListViewItem(parentptr, after,
                                                n.nodeValue());

        QDomNode child=n.firstChild();
        if(!child.isNull())
            traversenode(child,parentptr);

        n=n.nextSibling();
    }
}

int main (int argc, char *argv[])
{
    QApplication app(argc, argv);

    if(argc<2) {
        cout << "Usage: " << argv[0] << " <xmlfile>" << endl;
        exit(-1);
    }

    QDomDocument doc("mydocument");
    QFile f(argv[1]);
    if(f.open(IO_ReadOnly)) {
        QTextStream t(&f);
        QString s;

        while (!t.eof()) s.append(t.readLine());

        doc.setContent(s);
        f.close();
    } else {
        cout << "Couldn't open " << argv[1] << endl;
        exit(-1);
    }
}
```

```
    QListView lv(0,argv[1]);  
    lv.addColumn("Data");  
    lv.setSorting (-1);  
    lv.setRootIsDecorated(TRUE);  
  
    QListViewItem lvi(&lv,"XML Data")  
  
    QDomNode child=doc.firstChild();  
    child=child.nextSibling();  
  
    traversenode(child,&lvi);  
  
    app.setMainWidget(&lv);  
    lv.show();  
  
    return app.exec();  
}
```

28.4 小 结

用于 GUI 编程的 Qt C++ 类库为使用 C++ 构造 Linux 应用提供了丰富的窗口部件集合。Qt 的设计和实现做得非常好, 而且 Qt 可能会成为 Linux 的 C++ 程序员为 Linux 开发自由软件应用而选择的 GUI 库。C 程序员可能更偏向于使用第 27 章介绍的 GTK+ 库。通过支付一定的许可证费用 (参见 <http://www.trolltech.com>), Qt 也能用于商业应用。

即使你的 Linux 发布版本中包含了 Qt 的运行库和开发库 (作为单独设置选项, 你可能已经安装了这些软件包), 你也可能要访问 <http://www.trolltech.com>/ 查看网上的在线教程、PostScript 文档文件和最新的 Qt 发布版本。

第 29 章 使用 OpenGL 和 Mesa 进行 3D 图形编程

从头开始编写三维图形程序是一个非常耗时的过程。你不但要精通数学，尤其是线性代数，而且还要在精心调整优化处理和晦涩难懂的算法方面具有一定创造性以取得合理的性能。你需要完成大量的后台工作，并且开始在屏幕上编写出看得见的结果之前你还要构造许多函数。有几种图形库专门设计用来减轻程序员的上述负担，让他们跳过低层次编程的辛苦工作而直接开始有创造性的三维图形编程。无疑，使用最广泛的三维库是 OpenGL 库。

OpenGL API 由 Silicon Graphics, Inc(SGI)开发。最初设计它的目的是用于 Silicon Graphics 公司的高端图形工作站。它已经成为高质量三维图形的工业标准。在大多数现代操作系统上，比如 BeOS、MacOS、Windows 和 Linux，都有 OpenGL 的实现。

即使 OpenGL 有用于 Linux 的商业移植版本，但 Mesa 是 Linux 上最流行、使用最广泛的 OpenGL 实现。Mesa 是一种开放的、免费的软件，它实现了类似 OpenGL 的 API。因为它没有得到 Silicon Graphics 公司的许可证，所以不能叫作 OpenGL。取得许可证是一道非常昂贵的手续，所以只有大公司才能支付得起这笔费用。但是，Mesa 的确实现了 OpenGL API 提供的每种功能。即使本章在所有的例子中都使用了 Mesa API，我们还是常常把它叫作 OpenGL。

即使 Mesa 不是官方的 OpenGL 实现，但它在开放源代码界得到了高度评价。即使是 Silicon Graphics 公司也认为 Mesa 是一种良好的类 OpenGL API 实现。实际上，他们正在和几个伙伴共同制订 OpenGL 和 X Windows 下三维图形的标准，合作者中就有 Mesa 的创造者。这个计划叫作“Linux 上的 OpenGL 应用二进制接口”(OpenGL Application Binary Interface for Linux)，或者简称为 ABI。在撰写本书的时候，这项工作仍在进行中。如果想了解有关这个计划的更多信息，可以访问他们的 Web 网页 <http://oss.sgi.com/projects/ogl-sample/ABI/>。

29.1 需要为本章准备什么

在继续阅读本章的内容之前，请确认你的系统上已经安装了最新版本的 Mesa 软件包。如果没有 Mesa，查看一下本章末尾的小结，了解能从哪里获得它。在所有的 Linux 发布版本上都带有 Mesa 软件包，所以即使你愿意也能够编译 Mesa 库，也无需自己编译源代码。

不同的图形卡生产商提供了 Mesa 的几个修定版本。他们比一般的 Mesa 版本速度快得多。如果所使用的显卡品牌有这样一个版本的 Mesa，那么就应该使用它以得到系统最好的性能。安装这些软件包有时需要一点技巧，因为它们一般不是由 Linux 专家打包的。但是，很值得努力去尝试一下。

因为最近 XFree4.0、X Windows 都已经在其中包含了对三维加速的支持。有了这种支

持, 就更容易在 Linux 上取得良好的三维表现。X Windows 对三维图形的支持是以直接显示基础设施 (Direct Rendering Infrastructure, DRI) 的形式提供的。在最近才公开介绍这种技术, 所以到目前为止还没有多少 X Windows 驱动程序完全支持它。但事实上, Linux 上三维图形领域内的未来是充满希望的。

对于绝大多数 3D 图形应用来说, 你还需要一个单独的建模程序来创建 3D 形状, 而且在 OpenGL 程序中使用这些形状需要专门代码。给 3D 图像建模以及在 OpenGL 程序中使用它们超出了本章的范围。

29.2 使用 OpenGL

OpenGL API 非常复杂。但是, 只要使用很小一部分 API 就可能得到壮观的结果。你可以从只使用最基本的函数开始, 然后随着你对 API 知识的增长慢慢地加入新功能。

我们将在本章使用两个简单的示例程序来显示如何使用 OpenGL API。我们将使用某些辅助库来绘制简单的形状, 然后演示怎样在程序的控制下定位和旋转这些形状。我们还将学到如何使用光线效果和纹理映像以及如何改变视角。

示例程序使用了 OpenGL 工具库 (OpenGL Utilities Library, GLUT)。OpenGL API 本身既与操作系统无关, 也和设备无关。GLUT 库允许程序员以一种简单而且可移植的方式进行初始化 OpenGL、创建窗口等工作。

对于示例程序来说, 要保证检查 Mesa 发布版本的完整性。在 Mesa 的安装目录下有 3 个例子目录: book、demos 和 samples。这些目录包含了一些良好的示例程序, 你可以在学习用 OpenGL 进行三维编程时把它们作为参考。请确保在编译 Mesa 的同时也把这三个目录下的程序编译成二进制程序。在 Mesa 发布版本中的个别示例程序可能无法在你的显卡上运行, 但不必对此担心。

29.3 3D 图形编程

本章的示例程序展示了几种 OpenGL 编程技术。它们也足够简单, 可以作为程序教程。

29.3.1 orbits.c

第一个示例程序 orbits.c 位于 src/OpenGL/orbits 目录下。它使用 GLUT 函数 glutSolidSphere 绘制一个大行星和一个环绕它的小卫星。这个程序演示了以下几点:

- 为 OpenGL 图形创建窗口, 初始化 OpenGL
- 使用 GLUT 创建简单的三维物体
- 在 x, y, z 坐标系三维空间中的任意位置放置物体
- 沿 x-, y-, z-轴中的任意轴或所有轴旋转物体
- 启用材料属性为物体着色

- 启用深度检测使靠近观察者的着色物体能够正确覆盖掉被遮挡的物体
- 处理键盘事件
- 为获得动画效果，对 OpenGL 图形进行更新

这些操作是同时使用 OpenGL 核心应用程序编程接口 (API) 和 OpenGL 实用工具库完成的。所有的 OpenGL 实现都包含 OpenGL 实用工具库，因此本章中这个简单的示例程序应该可以很容易地移植到其他的操作系统和 OpenGL 实现上。

OpenGL 使用建模坐标系，其中 X 坐标向显示屏的右方为正向，Y 坐标向上为正向，而 z 坐标沿着朝向显示屏内部的方向为正向。原点（例如，在 $x=y=z=0$ 点）位于显示屏中心。

在程序运行过程中，单击任何键时（除了令程序终止的 `escape` 键或 `q` 键），程序将在不同的观察角度之间切换的同时，使图形的阴影部分光滑、平坦化。

图 29.1 显示了 Orbits 示例程序在默认的光滑阴影模式下的运行情况。注意在这张图中，背景颜色被改成白色。

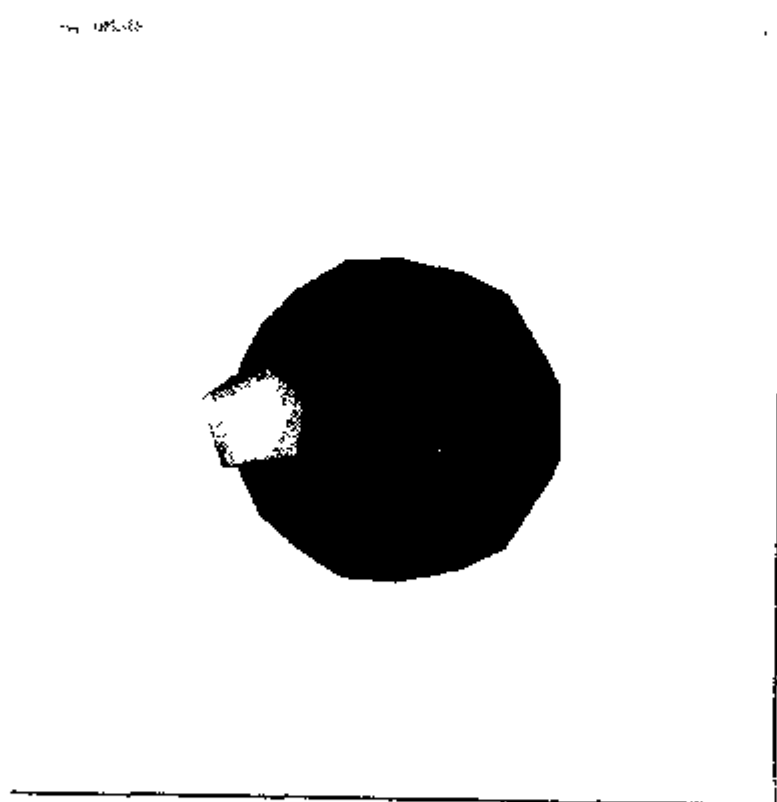


图 29.1 在光滑阴影模式下 Orbits 程序的运行情况

29.3.2 为 OpenGL 图形创建窗口并初始化 OpenGL

在本章中，将使用表 29.1 中的 OpenGL 实用工具库函数来对 OpenGL 和 GLUT 库进行初始化并创建一个画图的窗口：

表 29.1 OpenGL 实用工具库函数

函数	描述
<code>glutInit</code>	对 GLUT 和 OpenGL 进行初始化
<code>glutInitDisplayMode</code>	设置显示属性（在示例程序中设置的属性允许双缓冲、深度值排队和使用 RGB 颜色模式）

(续表)

函数	描述
<code>glutInitWindowSize</code>	设置主窗口的大小
<code>glutInitWindowPosition</code>	将主窗口放置在桌面上或者 X Windows 显示屏幕中
<code>glutCreateWindow</code>	完成实际的主窗口创建操作

示例程序中的初始化部分代码如下所示：

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
glutInitWindowSize(500, 500);
glutInitWindowPosition(100, 100);
glutCreateWindow(argv[0]);
```

在调用 `glutInitDisplayMode` 时，使用了表 29.2 中的常量。

表 29.2 调用 `glutInitDisplayMode` 时使用的常量

常量	描述
<code>GLUT_DOUBLE</code>	打开双缓冲区支持平滑动画
<code>GLUT_RGB</code>	指定使用 RGB 颜色表
<code>GLUT_DEPTH</code>	启用深度缓冲区来决定场景中一个物体什么时候遮掩了另外一个物体

29.3.3 使用 GLUT 创建简单的 3D 对象

有几个 GLUT 工具函数既可以创建线框图形又可以创建立体对象。在 OpenGL 中的对象是使用多边形而不是它们的真正数学形状来近似的。因此，像球体这样的对象其边界会显得有些粗糙而不够圆滑。一般说来，使用越多的多边形，对象就越接近实际的形状。另一方面，你使用的多边形越多，显示这个对象所需的计算机处理能力就越强。有近似部分的所有 GLUT 对象都有办法控制多边形的使用数量。通过改变这些参数，你就能调整你的程序让对象的形状尽可能的好，同时又能很快地显示。

创建一个球体

为了创建一个球体，可以使用下列函数中的任何一个：

```
void glutSolidSphere(GLdouble radius, GLint slices,
                    GLint stacks);
void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);
```

这里，`radius` 是球体的半径。参数 `slices` 是围绕 z 轴生成的平面带的个数。`slices` 的个数越多，球体就越圆。参数 `stacks` 是沿 z 轴方向平面带的个数。`stacks` 的值越大，球体就越圆。`slices` 和 `stacks` 的值越小，生成的图形便可以越快。一个球体以建模坐标系的原点为中心进行绘制或渲染（例如，当我们使用 OpenGL Matrix 作为建模模式时。后面将对此进行更多的说明）。

创建一个立方体

为了创建一个立方体，可以使用下列函数中的任何一个：

```
void glutSolidCube(GLdouble size);  
void glutWireCube(GLdouble size);
```

参数 `size` 表示立方体任何一边的长度。

创建一个锥体

为了创建一个锥体，可以使用下列函数中的任何一个：

```
void glutSolidCone(GLdouble base, GLdouble height,  
                  GLint slices, GLint stacks);  
void glutWireCone(GLdouble base, GLdouble height,  
                  GLint slices, GLint stacks);
```

参数 `base` 是圆锥体底面的半径。参数 `height` 是圆锥体的高。参数 `slices` 是围绕 `z` 轴绘制的平面带的个数。参数 `stacks` 是沿 `z` 轴绘制的平面带的个数。圆锥体底面的中心在建模坐标系坐标为(0,0,0)的点。圆锥体指向 `z` 轴方向。

创建一个圆环

为了创建一个圆环，可以使用下列函数中的任何一个：

```
void glutSolidTorus(GLdouble inner_radius,  
                   GLdouble outer_radius,  
                   GLint nsides, GLint rings);  
void glutWireTorus(GLdouble inner_radius,  
                   GLdouble outer_radius,  
                   GLint nsides, GLint rings);
```

参数 `inner_radius` 和 `outer_radius` 分别是圆环体的内径和外径。参数 `nsides` 为每一个径向的平面带个数。参数 `rings` 是径向带的个数。

29.3.4 使用 x-y-z 坐标在 3D 空间中放置对象

在将物体放置到三维世界之前，需要确保 OpenGL 处于建模模式。这可以通过调用下面的函数实现：

```
glMatrixMode(GL_MODELVIEW);
```

可以通过调用下面的函数将建模世界的坐标原点变换到坐标 `X,Y,Z`：

```
glTranslatef(GLfloat X, GLfloat Y, GLfloat Z);
```

这里有一个问题是必须记住我们将坐标原点移动到了什么地方。幸运的是，有这样的 OpenGL 实用函数可以将 OpenGL 的全部状态压入堆栈中，也可以从堆栈中弹出 OpenGL 的状态。这些函数如下所示：

```
glPushMatrix();
```

```
glPopMatrix();
```

实际上,通常将整个状态矩阵“matrix”(例如 OpenGL 引擎的状态)压入堆栈,对一个或多个物体执行绘制操作,然后将状态矩阵从堆栈中弹出。下面的例子显示了如何在 X=10.0,Y=0.0 和 Z=1.0 位置绘制一个半径为 1.0 的球体:

```
glPushMatrix();
{
    glColor4f(1.0, 0.0, 0.0, 1.0);    // make the sphere red
    glTranslatef(10.0, 0.0, 1.0);    // translate the model
                                     // coordinate system
    glutSolidSphere(1.0, 40, 40);    // Draw a very detailed
sphere
} glPopMatrix();
```

由于编程风格的原因,我将在压栈和出栈之间的任何操作都包含在一对花括号中,这样可以提高代码的可读性。

在本例中,还看到了一个对 glColor4f 函数的调用,它用来指定绘制的 RGB 颜色值(该值在 0.0 和 1.0 之间)。glColor4f 的第四个参数是 alpha 值。alpha 值用来设置物体透明度:alpha 为 0.0 则物体透明,但该值并不是特别的有用;alpha 为 1.0 则物体完全不透明。在示例程序中,试着在 display 的调用中将 alpha 值变为 0.5,这样你可以得到一个半透明物体,从而视线可以穿过该渲染物体。

29.3.5 沿着 x-、y-、z-中任一坐标轴或所有坐标轴旋转对象

旋转由用来绘制简单形体的 GLUT 实用工具创建的物体,比变换一个物体的 X, Y, Z 坐标要复杂一些。在示例程序的 display 函数中,绘制一个以 (0,0,0) 为中心的行星,和一个围绕该中央行星盘旋的小卫星。我们将首先来看围绕建模坐标系原点(例如, 0,0,0)旋转中央行星的程序代码:

```
// push matrix, draw central planet, then pop matrix:
glPushMatrix();
{
    glRotatef((GLfloat)planet_rotation_period,0.0, 1.0, 0.0);
    glColor4f(1.0, 0.0, 0.0, 1.0);
    glutSolidSphere(1.0, 10, 8);    // Draw the central planet
} glPopMatrix();
```

这里,使用变量 planet_rotation, 每次 OpenGL 场景被绘制时,该变量值都递增。该变量值在 0.0~360.0 之间,因为 glRotatef 函数的参数使用角度值,而不是弧度值。glRotatef 的后三个参数用来指定旋转轴。参数 (0.0,1.0,0.0) 指定旋转的角度围绕 y 轴的正方向。通过在 0.0~360.0 之间慢慢地改变 planet_rotation_period 的值,中央行星则慢慢地旋转。

卫星的放置和旋转则要复杂得多,因为既要变换它的位置,又要对其进行旋转。为了达到这一目的,示例程序代码中的 display 函数如下所示:

```
// push matrix, draw satellite, then pop matrix:
glPushMatrix();
{
    glRotatef((GLfloat)central_orbit_period, 0.0, 1.0, 0.0);
    glTranslatef(1.9, 0.0, 0.0);
    glRotatef((GLfloat)-satellite_rotation_period,
              0.0, 1.0, 0.0);
    glColor4f(0.0, 1.0, 0.0, 1.0);
    glutSolidSphere(0.2, 5, 4);    // Draw the orbiting satellite
}glPopMatrix();
```

起初，我们调用 `glRotatef` 来围绕原点旋转卫星——就像绘制中央卫星时所做的。认识到卫星在移动它之前以坐标原点为中心这一点很重要。在旋转坐标系之后，调用 `glTranslate` 将其移动到建模坐标系的 (1.9,0.0,0.0) 位置，最后再一次旋转坐标系来模拟卫星环绕行星的运动轨迹。

图 29.2 显示了示例程序 `Orbits` 在平滑阴影模式下的运行情况。图中屏幕的背景色被换色。

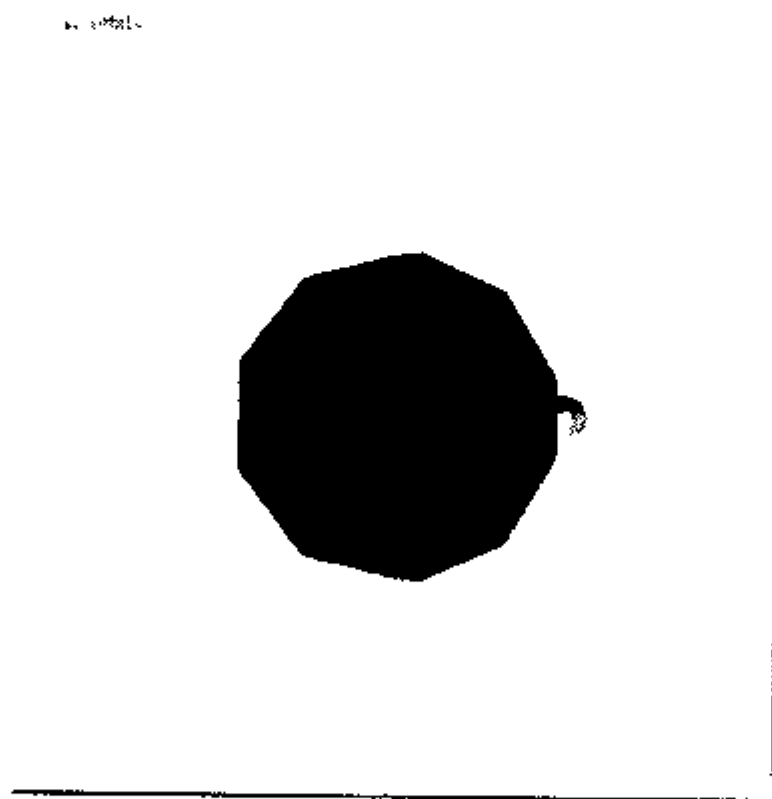


图 29.2 `Orbits` 程序在平滑阴影模式下运行

29.3.6 启用 Material 属性

示例程序的 `init` 函数用来设置 OpenGL 环境。下面的函数调用对 OpenGL 引擎进行配置以允许我们在后面使用绘制颜色：

```
glEnable(GL_COLOR_MATERIAL);
```

默认情况下，物体是平滑阴影的，`init` 也将 OpenGL 引擎配置为光滑阴影模式。当按下空格键时，示例程序在平滑和光滑阴影模式之间进行切换（同时也在三个视角之间进行切换）。下面的函数调用要求 OpenGL 为光滑阴影绘制模式：

```
glShadeModel(GL_SMOOTH);
```

也可以使用 `glClearColor` 来改变窗口的默认背景色。前三个参数是背景色的 RGB 值；最后一个参数用来指定背景的 alpha（透明）值。这里，为了使背景色接近黑色，我们设置的背景色的 R、G、B 分量都非常小。将背景的 alpha 值设置为 0，从而使背景完全透明。

```
glClearColor(0.1, 0.1, 0.1, 0.0);
```

在图 29.1 和图 29.2 中，通过将 `glClearColor` 函数的前 3 个参数都为 1.0，从而将背景设为白色。

29.3.7 启用深度测试

示例程序的 `init` 函数设置 OpenGL 环境使用深度测试来隐藏视线中被遮挡的物体。下面的函数调用配置 OpenGL 引擎以允许我们在后面使用深度线索：

```
glCullFace(GL_BACK);  
glEnable(GL_DEPTH_TEST);
```

除了调用这两个函数之外，当在主函数 `main` 中设置 GLUT 显示模式时，我们需要添加 `GLUT_DEPTH` 选项：

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
```

29.3.8 处理键盘事件

在 OpenGL 图形程序中，如果能够处理键盘输入是非常有用的。使用 GLUT 库函数，这很容易通过使用一个回调函数实现，每当键盘上有任意键按下时，该回调函数接受键盘的返回值 `key` 作为参数进行调用。示例程序中使用下面的回调函数：

```
void key_press_callback(unsigned char key, int x, int y)
```

在示例程序中，将使用第一个参数 `key`，而忽略最后的两个参数。我们通过调用下面的代码在主函数 `main` 中注册该回调函数：

```
glutKeyboardFunc(key_press_callback);
```

29.3.9 为获得动画效果更新 OpenGL 图形

在示例程序中，每当窗口需要重画时，使用 `display` 回调函数，该函数由 GLUT 库调用。`Display` 函数的原型定义如下：

```
void display(void)
```

通过调用下面的代码在主函数中注册该回调函数：

```
glutDisplayFunc(display);
```

在返回以前，`display` 所做的最后一件事是通过调用下面的代码请求一个立即重画事件：

```
glutPostRedisplay();
```

这将有效地使 OpenGL 引擎和 GLUT 事件处理器不断对动画进行更新。

29.3.10 Orbits 程序清单

在前面的几节中，已经看到了示例程序的大部分片段。在这一节中，将列出程序的整个文件清单，其中带有一些附加的注释。这个例子，如程序清单 29.1 所示，为调用 OpenGL/Mesa 和 OpenGL 实用工具函数，仅使用了单独一个 include 文件 glut.h。

程序清单 29.1 文件 orbits.c

```
#include <GL/glut.h>

void init(void)
{
    glClearColor(0.1, 0.1, 0.1, 0.0);
    glEnable(GL_COLOR_MATERIAL);
    glShadeModel(GL_SMOOTH);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_CULL_FACE);
    glCullFace(GL_BACK);
    glEnable(GL_DEPTH_TEST);
}

void display(void) {
    static int central_orbit_period = 0;
    static int planet_rotation_period = 0;
    static int satellite_rotation_period = 0;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // push matrix, draw central planet, then pop matrix:
    glPushMatrix();
    {
        glRotatef((GLfloat)planet_rotation_period, 0.0, 1.0, 0.0);
        glColor4f(1.0, 0.0, 0.0, 1.0);
        glutSolidSphere(1.0, 10, 8); // Draw the central planet
    } glPopMatrix();

    // push matrix, draw satellite, then pop matrix:
    glPushMatrix();
    {
        glRotatef((GLfloat)central_orbit_period, 0.0, 1.0, 0.0);
        glTranslatef(1.9, 0.0, 0.0);
        glRotatef((GLfloat)-satellite_rotation_period, 0.0, 1.0, 0.0);
        glColor4f(0.0, 1.0, 0.0, 1.0);
        glutSolidSphere(0.2, 5, 4); // Draw the orbiting satellite
    } glPopMatrix();

    glutSwapBuffers();
    central_orbit_period = (central_orbit_period + 2) % 360;
    planet_rotation_period = (planet_rotation_period + 1) % 360;
```

```
    satellite_rotation_period = (satellite_rotation_period + 6) % 360;
    glutPostRedisplay();
}

void reshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (GLfloat)w/(GLfloat)h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0, 0.0, 4.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}

void cycle_view() {
    static int count = 0;
    static int shade_flag = 0;
    if (++count > 2) {
        count = 0;
        shade_flag = 1 - shade_flag;
    }
    glLoadIdentity();
    switch (count)
    {
        case 0:
            gluLookAt(0.0, 0.0, 4.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
            break;
        case 1:
            gluLookAt(0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
            break;
        case 2:
            gluLookAt(0.0, 0.5, -3.3, 1.0, 0.0, 0.0, -0.7, 0.2, 0.4);
            break;
    }
    if (shade_flag == 0) glShadeModel(GL_SMOOTH);
    else glShadeModel(GL_FLAT);
}

void key_press_callback(unsigned char key, int x, int y) {
    switch (key)
    {
        case 27: /* escape character */
        case 'q':
        case 'Q':
            exit(1);
        default:
            cycle_view();
    }
}
```

```
        break;
    }
}

int main(int argc, char *argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutKeyboardFunc(key_press_callback);
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

29.4 纹理映像

为了让简单的多边形构成的形状看上去更真实，程序员经常使用一种称为纹理映像的技术。这意味着取得一幅位图并把它应用到多边形的表面。这样做能够让多边形显得比实际复杂了许多。通过在多个多边形上使用许多不同的纹理就能够让 Quake 那样的游戏获得一种真实而且复杂的显示环境，同时实际又足以简单到让计算机的处理器能够很快地计算并显示出每一帧画面。

这里的例子会生成一个立方体，在它的每个面上带有简单的纹理。这个立方体没有完全封闭，而是让它的各个面稍稍有些分开，这样我们就能够在立方体旋转的时候看到它的内部。纹理是通过编程产生的。但是，我们把一幅位图加载进来作为纹理的操作也同样简单。这个例子以 Chris Halsall 提供的可自由获得的示例程序为基础。

这个程序的大部分对你来说都很熟悉。但是，我们会仔细讨论一些新用到的函数。在你需要对任何 API 进一步的帮助时可以参考 OpenGL 的在线文档。

29.4.1 用纹理面产生立方体

这个示例程序没有使用 GLUT 来创建它的对象。相反，它使用 OpenGL API 来直接创建用于立方体的多边形。这样做能够更好地控制立方体每个面的颜色和纹理设置。

要做的第一件事是告诉 OpenGL 我们将要开始为一个对象定义多边形。这通过下面的 OpenGL 函数来完成：

```
glBegin(GL_QUADS);
```

这条语句告诉 OpenGL 每个多边形都由四个点组成。

接下来定义立方体的每个面，一次定义一个。在这段代码中，定义了立方体的一个面：

```
glNormal3f( 0.0f,0.0f,-1.0f); glColor4f(0.2,0.9,0.2,.5);  
glTexCoord2f(0.995f,0.005f);glVertex3f(-1.0f,-1.0f,-1.3f);  
glTexCoord2f(2.995f,2.995f);glVertex3f(-1.0f,1.0f,-1.3f);  
glTexCoord2f(0.005f,0.995f);glVertex3f(1.0f,-1.0f,-1.3f);  
glTexCoord2f(0.005f,0.005f);glVertex3f(1.0f,-1.0f,-1.3f);
```

我们需要首先为每个面定义的是多边形的表面应该面向的方向。这称为法向量，它用来计算光线怎样从表面进行反射。使用 `glNormal3f` API 调用设置它。

接下来定义表面应该具有的颜色和透明度。在示例代码的第一行设置颜色。然后加上每个顶点或者说端点，它们应该包括在多边形中。注意，最后一个顶点和第一个顶点相同，因此构成了封闭的多边形。另外，对于每个顶点都要设置怎样将纹理应用到表面上。

如果为多边形的某些顶点指定了不同的颜色，那么 OpenGL 将自动地产生梯度填充，平滑地从一种颜色过渡到另外一种颜色。我们对立方体一面所做的设置如下：

```
glNormal3f( 0.0f, 0.0f, 1.0f);  
glColor4f( 0.9f, 0.2f, 0.2f, 0.5f);  
glTexCoord2f( 0.005f, 0.005f); glVertex3f(-1.0f, -1.0f, 1.3f);  
glColor4f(0.2f, 0.9f, 0.2f, 0.5f,);  
glTexCoord2f( 0.995f, 0.005f); glVertex3f( 1.0f, -1.0f, 1.3f);  
glColor4f(0.2f, 0.2f, 0.9f, 0.5f,);  
glTexCoord2f( 0.995f, 0.995f); glVertex3f( 1.0f, 1.0f, 1.3f);  
glColor4f( 0.1f, 0.1f, 0.1f, 0.5f);  
glTexCoord2f( 0.005f, 0.995f); glVertex3f(-1.0f, 1.0f, 1.3f);
```

最后，一旦定义完毕构成对象的所有多边形，就需要告诉 OpenGL 这个对象完成了：

```
glEnd();
```

这个程序使用的纹理是由程序构造的，纹理是由一些方块和圆圈构成的图案。函数 `outBuildTextures` 构造出纹理。它调用了 GLU 的一些功能产生了 `mipmap`。它们可以当作是一系列不同分辨率的纹理映像图组成的层次结构。随着观察者逐步接近带有纹理的对象，OpenGL 会使用插值为映像创建更高分辨率的纹理。通过这种方法，你就不会在对象很远的时候为了显示其不必要的细节而浪费处理器的处理能力去显示所定义的更高分辨率的映像。

29.4.2 创建纹理映像

当创建纹理映像时要做的第一件事情就是为使用纹理初始化 OpenGL。这通过下面的语句来完成：

```
glGenTextures(1, &Texture_ID);  
glBindTexture(GL_TEXTURE_2D,Texture_ID);
```

第一个函数创建了纹理索引，可以用它来保存有关每个纹理的信息。然后 `glBindTexture` 定义了这是一个二维的纹理。在这之后，程序使用图案填充纹理数组。随后把这个数组传递给构造 `mipmaps` 的 GLU 函数：

```
gluBuild2DMipmaps(GL_TEXTURE_2D, 4, 128, 128, GL_RGBA,
                  GL_UNSIGNED_BYTE, (void *)tex);
```

就需要这么多工作。其余的工作由 OpenGL 自动完成。

你可以和这个程序进行交互。可以用 Page Up 和 Page Down 键改变观察者和立方体之间的距离。你可以使用光标改变立方体旋转的方向和速度。ESC 键用于从程序中退出。图 29.3 显示了 Cube 程序正在执行时的一个情景。

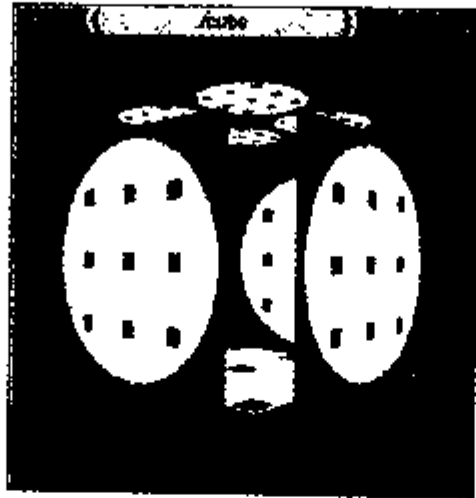


图 29.3 Cube 程序

29.4.3 立方体程序清单

已经介绍过了 Cube 程序的几个部分，现在是看看整个程序的时候了。程序清单 29.2 给出了完整的 cube.c 文件。

程序清单 29.2 文件 cube.c

```
#include <stdio.h>
#include <GL/glut.h>

int Texture_ID;
int Window_Width = 300;
int Window_Height = 300;

float X_Rot = 0.9f;
float Y_Rot = 0.0f;
float X_Speed = 0.0f;
float Y_Speed = 0.5f;
float Z_Off = -5.0f;

void cbRenderScene(void)
{
    glEnable(GL_TEXTURE_2D);
    glEnable(GL_LIGHTING);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glEnable(GL_DEPTH_TEST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST_MIPMAP_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
```

```

        GL_NEAREST);

glMatrixMode(GL_MODELVIEW) ;
glLoadIdentity();
glTranslatef(0.0f,0.0f,Z_Off);
glRotatef(X_Rot,1.0f,0.0f,0.0f);
glRotatef(Y_Rot, 0.0f, 1.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glBegin(GL_QUADS);
glNormal3f( 0.0f,-1.0f, 0.0f); glColor4f(0.9,0.2,0.2,.75);
glTexCoord2f(0.800f, 0.800f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(0.200f, 0.800f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(0.200f, 0.200f); glVertex3f( 1.0f, -1.0f,  1.0f);
glTexCoord2f(0.800f, 0.200f); glVertex3f(-1.0f, -1.0f,  1.0f);

glNormal3f( 0.0f, 1.0f, 0.0f); glColor4f(0.5,0.5,0.5,.5);
glTexCoord2f(0.005f, 1.995f) glVertex3f(-1.0f,  1.3f, -1.0f);
glTexCoord2f(0.005f, 0.005f) glVertex3f(-1.0f,  1.3f,  1.0f);
glTexCoord2f(1.995f, 0.005f) glVertex3f( 1.0f,  1.3f,  1.0f);
glTexCoord2f(1.995f, 1.995f) glVertex3f( 1.0f,  1.3f, -1.0f);

glNormal3f( 0.0f, 0.0f,-1.0f); glColor4f(0.2,0.9,0.2,.5);
glTexCoord2f(0.995f, 0.005f) glVertex3f(-1.0f, -1.0f, -1.3f);
glTexCoord2f(2.995f, 2.995f) glVertex3f(-1.0f,  1.0f, -1.3f);
glTexCoord2f(0.005f, 0.995f) glVertex3f( 1.0f,  1.0f, -1.3f);
glTexCoord2f(0.005f, 0.005f) glVertex3f( 1.0f, -1.0f, -1.3f);

glNormal3f( 1.0f, 0.0f, 0.0f); glColor4f(0.2,0.2,0.9,.25) ;
glTexCoord2f(0.995f, 0.005f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(0.995f, 0.995f); glVertex3f( 1.0f,  1.0f, -1.0f);
glTexCoord2f(0.005f, 0.995f); glVertex3f( 1.0f,  1.0f,  1.0f);
glTexCoord2f(0.005f, 0.005f); glVertex3f( 1.0f, -1.0f,  1.0f);

glNormal3f( 0.0f, 0.0f, 1.0f);
glColor4f(0.9f, 0.2f, 0.2f, 0.5f);glTexCoord2f( 0.005f, 0.005f);
glVertex3f(-1.0f, -1.0f,  1.3f);
glColor4f(0.2f, 0.9f, 0.2f, 0.5f); glTexCoord2f( 0.995f, 0.005f);
glVertex3f( 1.0f, -1.0f,  1.3f);
glColor4f(0.2f, 0.2f, 0.9f, 0.5f); glTexCoord2f( 0.995f, 0.995f);
glVertex3f( 1.0f,  1.0f,  1.3f);
glColor4f(0.1f, 0.1f, 0.1f, 0.5f);
glTexCoord2f( 0.005f, 0.995f);
glVertex3f(-1.0f,  1.0f,  1.3f);

glNormal3f(-1.0f, 0.0f, 0.0f); glColor4f(0.9,0.9,0.2,0.0);
glTexCoord2f(0.005f, 0.005f); glVertex3f(-1.3f, -1.0f, -1.0f);
glTexCoord2f(0.995f, 0.005f); glVertex3f(-1.3f, -1.0f,  1.0f);
glTexCoord2f(0.995f, 0.995f); glVertex3f(-1.3f,  1.0f,  1.0f);
glTexCoord2f(0.005f, 0.995f); glVertex3f(-1.3f,  1.0f, -1.0f);

```

```

    glEnd();
    glutSwapBuffers();

    X_Rot+=X_Speed;
    Y_Rot+=Y_Speed;
}

void cbKeyPressed(unsigned char key, int x, int y)
{
    if(key == 27)    exit;
}

void cb8specialKeyPressed(int key, int x, int y)
{
    switch (key) {
    case GLUT_KEY_PAGE_UP:
        Z_Off -= 0.05f;    break;
    case GLUT_KEY_PAGE_DOWN:
        Z_Off += 0.05f;    break;
    case GLUT_KEY_UP:
        X_Speed -= 0.01f; break;
    case GLUT_KEY_DOWN:
        X_Speed += 0.01f; break;
    case GLUT_KEY_LEFT:
        Y_Speed -= 0.01f, break;
    case GLUT_KEY_RIGHT:
        Y_Speed += 0.01f, break;
    }
}

void ourBuildTextures(void)
{
    GLenum glerr;
    GLubyte tex[128][128][4];
    int x,y,t;
    int hole_size = 3300; //~ == 57.45 ^ 2.

    glGenTextures(1,&Texture_ID);
    glBindTexture(GL_TEXTURE_2D,Texture_ID);
    for(y=0;y<128;y++) {
        for(x=0;x<128;x++) {
            if ( ( (x+4)%32 < 8 && ( (y+4)%32 < 8)) ) {
                tex[x][y][0]=tex[x][y][1]=0; tex[x][y][2]=120;
            }
            else tex[x][y][0]=tex[x][y][1]=tex[x][y][2]=240;
            t = (x-64)*(x-64) + (y-64)*(y-64) ;
            if ( t < hole_size)
                tex[x][y][3]=255;
            else if (t < hole_size + 100)
                tex[x][y][3]=128;
        }
    }
}

```

```

        else tex[x][y][3]=0;
    )
}
if ((gluerr=gluBuild2DMipmaps(GL_TEXTURE_2D, 4, 128, 128,
    GL_RGBA, GL_UNSIGNED_BYTE, (void *)tex))) {
    fprintf(stderr, "GLULib%s\n", gluErrorString(gluerr)) ;
    exit(-1);
}
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT;;
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT;;
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
)
void cbResizeScene(int Width, int Height)
{
    if (Height == 0) Height = 1;
    glViewport(0, 0, Width, Height);
    glMatrixMode(GL_PROJECTION) ;
    glLoadIdentity() ;
    gluPerspective(45.0f, (GLfloat)Width/(GLfloat)Height,
        0.1f, 100.0f);
    glMatrixMode(GL_MODELVIEW);
    Window_Width = Width;
    Window_Height = Height;
}
void ourInit(int Width, int Height)
{
    ourBuildTextures();

    glClearColor(0.1f, 0.1f, 0.1f, 0.0f);
    glClearDepth(1.0);
    glDepthFunc(GL_LESS);
    glShadeModel(GL_SMOOTH);

    cbResizeScene(Width, Height) ;

    glLightfv(GL_LIGHT1, GL_POSITION, (float []){ 2.0f, 2.0f, 0.0f,
        1.0f });
    glLightfv(GL_LIGHT1, GL_AMBIENT, (float []){ 0.1f, 0.1f, 0.1f,
        1.0f });
    glLightfv(GL_LIGHT1, GL_DIFFUSE, (float []){ 1.2f, 1.2f, 1.2f,
        1.0f });
    glEnable (GL_LIGHT1);

    glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);
    glEnable(GL_COLOR_MATERIAL);
}

int main(int argc, char *argv[])
{
    glutInit(&argc, argv);

```



```
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);  
glutInitWindowSize(Window_Width, Window_Height);  
glutCreateWindow(argv[0]);  
glutDisplayFunc(&cbRenderScene);  
glutIdleFunc(&cbRenderScene);  
glutReshapeFunc(&cbResizeScene);  
glutKeyboardFunc(&cbKeyPressed);  
glutSpecialFunc(&cbSpecialKeyPressed);  
ourInit(Window_Width, Window_Height);  
glutMainLoop();  
return 0;  
}
```

29.5 小 结

本章中的示例程序展示了如何产生简单几何对象的动画效果。这些程序可以作为编写简单三维游戏的基础，这些三维游戏只使用在 OpenGL 工具库中定义的三维形状（比如球体、立方体、圆锥体等）。

在 Web 上有许多很好的 OpenGL 资源。首先也是最重要的是 <http://www.opengl.org/>。这个站点上有许多文档、教程和链接到其他出色的 OpenGL 站点的链接。Neon Helium Productions 在它自己的站点上有些出色的 OpenGL 教程。你可以在 <http://nehe.gamedev.net/opengl.asp> 上找到这些教程。最后，如果你想了解有关 Mesa 的更多信息，可以查询它 Web 站点 <http://mesa3d.sourceforge.net/>。

第6部分 特殊编程技术

第30章 使用 GNU Bash 进行 Shell 编程

1993 年，当我刚刚成为一个 Linux 用户的头两天，我就编写了我的第一个 bash 脚本。回想起来，第一次尝试用脚本自动备份却写得很乱。然而，在过去 8 年的维护过程中，我持续不断地对它进行了精雕细琢，因为即使这个程序非常短小，但它却能可靠地发挥自己的作用。为了做对比，作为 OpenLinux 一部分的 LISA (Linux 安装和系统管理，Linux Installation and System Administration) 工具是一组功能完善、强健而且可靠的 bash shell 脚本，从功能上看，它们能够和编译好的代码相媲美。这段小故事展示了 shell 脚本在 Linux 上的应用有多么普遍。本章将为你在 bash 下进行基本的 shell 编程打下坚实的基础。

30.1 为何使用 bash

GNU 的 bash (Bash Again Shell，这个名字有双关意义，它表示了对 Bourne Shell 和它的作者 Steven Bourne 的敬意) 是 Linux 上默认使用的 shell。它最初是由 Brian Fox 编写的，而现在由 Chet Ramey 负责维护。从一个终端用户的角度来看，bash 最流行的特性是它丰富的命令行编辑功能和它的作业控制能力。而从一个程序员的角度来看，bash 最大的优点是它的可定制性以及它提供的完整的编程环境，包括函数定义、整数运算和出奇完备的 I/O 接口。正如你对一种 Linux 工具所期望的那样，bash 包含所有流行的 shell——Bourne、Korn 和 C——所具备的要素，同时还带有它自己的一些创新之处。在编写本书的时候，bash 的当前版本是 2.0.4。但是，大多数 Linux 发布版本（即使它们也常常包含版本 2）默认却使用了 1.14 这个测试版本。本书的讨论将集中在版本 2 上，同时也介绍较早版本所没有的功能，因为版本 2 比版本 1 更接近 POSIX 兼容性标准。

30.2 bash 基础知识

本章的内容假定读者作为一个终端用户已经能够轻松地使用 bash 的一般功能，所以将从 shell 程序员的角度集中讨论 bash 的特性。但是，通过回顾，在前两节让读者复习一下 bash 的通配符和它的特殊字符。

30.2.1 通配符

bash 的通配符有“*”、“?”和集合运算符[SET]和[!SET]。“*”匹配任何字符串，而“?”匹配任何单个字符。例如，假设你有一个目录包括下列文件：

```
$ ls
zeisstopnm    zic2xpm      zipgrep      zipsplit     znew
zcmp          zforce       zip          zipinfo      zless
zdiff         zgrep        zipcloak     zipnote      zmore
```

命令 `ls zi*` 将匹配 `zic2xpm`、`zip`、`zipcloak`、`zipgrep`、`zipinfo` 和 `zipsplit`，但是 `ls zi?` 只匹配 `zip`。

集合运算符允许你使用一个连续范围的字符集合或者一个断续的字符集合（[!SET]）作为通配符。要指定一个字符集合范围，可以在两个字符之间使用连字符。例如，集合 `[a-o]` 包括所有从 `a~o` 的小写字母。使用逗号来表示断续的集合，比如从 `a~h` 和从 `w~z` 的这两段字符范围。用 bash 的集合记法，上述范围的字符就可以用 `[a-h, w-z]` 来表示。如果你只想表示一些单个字符，比如所有的英语元音字母，可以逐一列举这些字母，用 `[aeiou]` 这一记法表示它们。在集合前的“!”前缀表示包括除了集合包含的字符以外的所有字符组成的集合。因此，要表示所有辅音字母，最简单的办法是写成 `[!aeiou]`。

下面的例子展示了集合记法的使用以及如何把通配符和集合记法联合起来使用。为了列出上述目录中所有的文件名第二个字母为元音字母的文件，可以使用命令：

```
$ ls z[aeiou]*
```

这将匹配 `zeisstopnm`、`zip`、`zipgrep`、`zipnote`、`zic2xpm`、`zipcloak`、`zipinfo` 和 `zipsplit`。另一方面，列出文件名的第二个字母为辅音字母的文件，则使用命令：

```
$ ls z[!aeiou]*
```

这个例子匹配 `zcmp`、`zdiff`、`zforce`、`zgrep`、`zless`、`zmore` 和 `znew`。命令 `ls *[0-9]*` 匹配至少含有 0~9 之间一个数字的文件名，在本例中，它将匹配 `zic2xpm`。

30.2.2 花括号展开式

花括弧展开式是利用通配符查找文件名的一个更通用的方法。文件名扩展是指将通配表达式扩展为一个或者更多个通配表达式所匹配的文件名的方式。

使用花括弧展开式的基本格式是：

```
[前导字符串]{字符串 1[, {字符串 2[, ...]]}[后继字符串]
```

每个花括弧中的字符串将与前导字符串和后继字符串匹配，例如下面的语句：

```
$echo c{ar, at, an, on}s
```

结果为：

```
cars cats cans
```

因为花括弧展开式可以嵌套，上面的命令可以改写为下面的这个命令，结果是获得同样的输出：

```
$ echo c{a{r, t, n}}s
```

30.2.3 特殊字符

通配符和 set 操作符是 bash 的特殊字符（对 bash 有特殊含义的字符）的例子。表 30.1 列出了所有的特殊字符，并且对它们做简要描述。

表 30.1 bash 的特殊字符

字符	描述
<	输入重定向
>	输出重定向
(子 shell 开始
)	子 shell 结束
	管道
\	引用（转换）下面的字符
&	在后台执行命令
{	命令块起始
}	命令块结束
~	根目录
,	命令替换
;	命令分隔符
#	注释
'	强引用
"	弱引用
\$	变量表达式
*	字符串通配符
?	单个字符通配符

你可能熟悉输入输出重定向。虽然我将在本章的后面讨论子 shell，要注意到在（和）之间的所有命令是在子 shell 中执行的，这一点非常重要。子 shell 继承父 shell 的一些环境变量，但不是全部。这个特点和块里的命令不同（块由{和}括起来），这些命令由当前 shell 来执行，从而保留了所有的当前环境。

命令分隔符；允许你在一行里面执行多个 bash 命令。更重要的是，它是 POSIX 指定的命令中止符。

注释字符#使得 bash 忽略从该字符到行末的所有字符。强引用字符'和弱引用字符"之间的区别在于，强引用迫使 bash 按字面来解释所有特殊字符，而弱引用仅使得某些特殊字符不被解释为特殊字符。

30.3 使用 bash 变量

正如你所预料的那样，bash 有变量。通常它们是字符串，但是 bash 也有特殊变量来处理数目（整数）值。为了给一个变量赋值，使用下面的语法：

```
varname = value
```

为了获得一个变量的值，你可以使用下面两种格式：

```
$varname  
${varname}
```

第二种格式，`${varname}`，是更普遍的格式但键入时麻烦一些。但是，你必须使用它来区分尾随的字母、数字或下划线。例如，假定你有一个变量叫 MYNAME，你想显示它的值，并在后面跟一个下划线字符。你可以试试下面命令：

```
$ echo $MYNAME_
```

但是这个尝试会失败，因为 bash 将尝试打印变量 MYNAME_ 的值，而不是 MYNAME。在这种情况下，你必须使用第二种语法，如下：

```
$ echo ${MYNAME}_
```

程序清单 30.1 描述了这种细微差异。

程序清单 30.1 变量语法

```
#!/bin/bash  
# varref.sh - Variable syntax  
  
MYNAME='Kurt Wall'  
echo '${MYNAME} yields: ' ${MYNAME}_  
echo '$MYNAME_ yields: ' $MYNAME_
```

最后两行的强引用字符防止 bash 扩展变量。正如下面你所看到的脚本输出，\$MYNAME_ 是空的没有打印输出，但是 \${MYNAME}_ 产生预想的打印输出。

```
$ ./varref.sh  
$ {MYNAME}_ yields: Kurt Wall_  
$ MYNAME_ yields:
```

除了用户定义的变量之外，bash 还带有大量内置或者预定义的变量。这些变量中有许多是环境变量，比如 \$BASH_VERSION 或者 \$DIRSTACK。例如，在 shell 命令行发出 set 命令时，应该产生类似下面的输出：

```
$ set  
BASH=/bin/bash2  
BASH_VERSINFO=([0]="2" [1]="03" [2]="0" [3]="1" [4]="release"  
               [5]="i386-caldera-linux-gnu")  
BASH_VERSION='2.03.0(1)-release'
```

```
COLUMNS=80
DIRSTACK=()
display=:0.0
EUID=500
GROUPS=()
HELPPATH=/usr/openwin/help
HISTFILE=/home/kwall/.bash_history
HISTFILESIZE=100
HISTSIZE=100
HOME=/home/kwall
HOSTTYPE=i386
IFS='
'
JAVA_HOME=/usr/java
KDEDIR=/opt/kde
LESSCHARSET=latin1
LINES=50
LOGNAME=kwall
MACHTYPE=i386-caldera-linux-gnu
MAIL=/var/spool/mail/kwall
MAILCHECK=60
OLDPWD=/home/kwall/lpu2/32/tmp
OPENWINHOME=/usr/openwin
OPTERR=1
OPTIND=1
OSTYPE=linux-gnu
PAGER=less
PATH=/home/kwall/bin:/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin
:/opt/bin:/opt/tex/bin:/opt/kde/bin:/usr/java/bin
PIPESTATUS=({0}="0")
PPID=948
PS1='[\u@\h \w]\$ '
PS2='> '
PS4='+ '
PWD=/home/kwall/lpu2/32
SHELL=/bin/bash
SHELLOPTS=braceexpand:hashall:histexpand:monitor:history:
interactive-comments:emacs
SHLVL =2
TERM=xterm-color
UID=500
USER=kwall
_=set
_ETC_PROFILE=1
```

另一个预定义变量的子集是包含命令行参数的位置参数，如果有命令行参数的话，这些参数将传给脚本（shell 函数，后面将提到，也接受参数）。位置参数“\$0”包含脚本名称：

实际参数从“\$1”开始（如果参数超过 9 个，必须使用\${}语法来获得参数值）。传给每个脚本或者函数的位置参数是局部和只读的。但是，其他变量默认是全局变量，除非你用 local 关键字声明它们。

bash 还有 3 个位置参数，“\$#”，“\$@”，以及“\$*”。“\$#”表示传给脚本或者函数的位置参数的个数（不包括“\$0”）。例如，shell 命令 echo one two three 有 4 个位置参数，在命令行上键入的命令名（echo）和 3 个参量（one、two 和 three）。

“\$*”是所有位置参数（除了“\$0”）的列表，其形式是一个单个字符串，串中每个参数由第 1 个字符串 IFS（内部域分隔符——bash 的另一个环境变量）分隔。它通常是个空字符串。另一方面，“\$@”是所有位置参数被分别表示为双引号中的 N^①个字符串。

“\$*”和“\$@”之间有什么区别呢？为什么有这种区别呢？它们之间的不同允许你用两种方法来处理命令行参数。第一种格式为“\$*”，因为它是一个单个字符，所以可以不需要很多 shell 代码来显示它，相比之下更加灵活。另一种格式为“\$@”，它允许你独立处理每个参数，因为它的值是 N 个分离参数。程序清单 30.2 描述了“\$#”和“\$@”之间的差别。

程序清单 30.2 位置参数

```
#!/bin/bash2
# posparm.sh - Using positional parameters

function cntparm
{
    echo -e "inside cntparm: $# parms: $*\n"
}

cntparm "$*"
cntparm "$@"

echo -e "outside cntparm: $*\n"
echo -e "outside cntparm: $@\n"
```

这个脚本的输出如下：

```
$ ./posparm.sh Kurt Roland Wall
inside cntparm 1 parms: Kurt Roland Wall

inside cntparm 3 parms: Kurt Roland Wall

outside cntparm Kurt Roland Wall

outside cntparm Kurt Roland Wall
```

两次调用 shell 函数 cntparm 显示了位置参数“\$*”和“\$@”的实际差异。现在暂时把函数定义为一个黑盒子。第一个调用使用“\$*”，它把位置参数作为单个字符串，所以 cntparm 报告说只有一个参数，正如在第一行输出所显示的那样。但是，第二次调用 cntparm，把脚本的命令行参数当作了三个字符串，所以 cntparm 报告了三个参数，正如在第二行输出所

^① 译者注：N 为参数个数。

显示的那样。但是在打印的时候，参数的外观没有任何区别，最后两行输出清楚地表明了这一点。

`echo` 命令的 `-e` 选项强行要求它把字符串序列 `\n` 当作一个新行（第 8、14 和 15 行）。如果你没有使用 `-e` 选项，就不要用 `\n` 来产生一个新行，因为 `echo` 会自动地在输出后面加上一个 `\n`。

30.4 使用 bash 操作符

我在讨论其他主题的过程中引入了很多 `bash` 操作符。在本节里，我将让你熟悉 `bash` 的字符串和模式匹配操作符，后面的章节里将频繁使用这些操作符，所以现在讨论它们将使后面章节的处理变得容易。

30.4.1 字符串操作符

字符串操作符，在 `bash` 文档里也称为替换操作符，测试一个变量是否没有设置值或者是空。表 30.2 列出了这些操作符以及操作符功能的简明描述。

表 30.2 `bash` 字符串操作符

操作符	功能
<code>\${var:-word}</code>	如果 <code>var</code> 存在且不为空，返回它的值，否则返回 <code>word</code>
<code>\${var:=word}</code>	如果 <code>var</code> 存在且不为空，返回它的值，否则将 <code>var</code> 设为 <code>word</code> ，然后返回它的值
<code>\${var:+word}</code>	如果 <code>var</code> 存在且不为空，返回 <code>word</code> ，否则返回空
<code>\${var:?message}</code>	如果 <code>var</code> 存在且不为空，返回它的值，否则显示 “ <code>bash2: \$var:\$message</code> ” 然后退出当前命令或者脚本
<code>\${var:offset[:length]}</code>	从 <code>offset</code> 标明的地方开始返回 <code>var</code> 的一个长为 <code>length</code> 的子串。如果没有给出 <code>length</code> ，从 <code>offset</code> 处开始的串将都被显示

为了描述方便，考虑一个 `shell` 变量 `STATUS` 初始化为 `Rich, famous, handsome`。使用表 30.2 中的头 4 个字符串操作符对 `STATUS` 进行操作，结果如下：

```
$ STATUS = "Rich, famous, handsome."
$ echo ${STATUS:-~Poor, unknown, ugly.}
Rich, famous, handsome.
$ echo ${STATUS:=Poor, unknown, ugly.}
Rich, famous, handsome.
$ echo ${STATUS:+Poor, unknown, ugly}
Poor, unknown, ugly.
$ echo ${STATUS:?Poor, unknown, ugly.}
Rich, famous, handsome.
```

现在，使用 `unset` 命令从环境中删除 `STATUS` 的定义，然后执行同样的命令，输出如下：


```

$ unset STATUS
$ echo ${STATUS:-Poor, unknown, ugly.}
Poor, unknown, ugly.
$ echo ${STATUS:=Poor, unknown, ugly.}
Poor, unknown, ugly.
$ echo ${STATUS:+Poor, unknown, ugly.}
Poor, unknown, ugly.
$ unset STATUS
$ echo ${STATUS:?Poor, unknown, ugly.}
bash2: STATUS: Poor, unknown, ugly.

```

第二次使用 `unset STATUS` 命令是必要的，因为第三个命令 `echo${STATUS:+Poor, unknown vgly.}` 执行后，重新设置 `STATUS` 为 `Poor, unknown, vgly.`

表 30.2 底部的子串操作符对获得子串特别有用。考虑一个值为 `Bilbo_the_Hobbit` 的变量 `foo`。表达式 `${foo:7}` 返回 `he_Hobbit`，而 `${foo:7:5}` 返回 `he_Ho`。这两个操作符在处理已知的固定格式的数据时最为有用（例如 `ls` 命令的输出）。但是要注意，`ls` 命令的输出格式在不同的 UNIX 操作系统下有很大不同，所以这种类型的 shell 代码将不能移植。

30.4.2 模式匹配操作符

模式匹配操作符对自由格式的字符串或者变长的以固定字符为边界的记录进行处理是最为有用的。`$PATH` 环境变量就是一个例子。虽然它可以很长，但是每个目录都以冒号为界。表 30.3 列出了 `bash` 的模式匹配操作符和它们的功能。

表 30.3 bash 模式匹配操作符

操作符	功能
<code>\${var#pattern}</code>	从 <code>var</code> 头部开始，删除和 <code>pattern</code> 匹配的最短模式串，然后返回剩余串
<code>\${var##pattern}</code>	从 <code>var</code> 头部开始，删除和 <code>pattern</code> 匹配的最长模式串，然后返回剩余串
<code>\${var%pattern}</code>	从 <code>var</code> 尾部开始，删除和 <code>pattern</code> 匹配的最短模式串，然后返回剩余串
<code>\${var%%pattern}</code>	从 <code>var</code> 尾部开始，删除和 <code>pattern</code> 匹配的最长模式串，然后返回剩余串
<code>\${var/pattern/string}</code>	用 <code>string</code> 替换 <code>var</code> 中和 <code>pattern</code> 匹配的最长模式串。仅替换第一个匹配的串。这个操作仅在 <code>bash 2.0</code> 版本及以上版本中提供
<code>\${var/pattern/string}</code>	用 <code>string</code> 替换 <code>var</code> 中和 <code>pattern</code> 匹配的最长模式串。替换所有匹配的串。这个操作仅在 <code>bash 2.0</code> 版本及以上版本中提供

经典的 `bash` 模式匹配操作符用法是操纵文件名和路径名。例如，假定你有一个 shell 变量叫 `MYFILE`，值为 `/usr/src/linux/Documentation/ide.txt`（内核 IDE 硬盘驱动程序的文档）。使用 `/*` 和 `*/` 作为模式，你可以模仿 `dirname` 和 `basename` 命令操作的行为，输出如程序清单 30.3 所示。

注意：`dirname` 命令从传入的文件名参数中删除非目录的后缀，将结果打印到标准输出。相反，`basename` 命令从文件名中删除命令前缀。作为 GNU shell 工具的一部分，`dirname` 和 `basename` 的手册页面比较难找到，如果你需要关于它们的

进一步信息，你将不得不阅读这些帮助信息页面。要引用它们的帮助手册内容，注意这些文档现在是权威性的资源。

程序清单 30.3 模式匹配操作符

```
#!/bin/bash2

# pattern.sh - Demonstrate pattern matching operators

MYFILE=/usr/src/linux/Documentation/ide.txt

echo "${MYFILE##*/}      =" "${myfile##*/}"
echo "basename $MYFILE =" $(basename $MYFILE)

echo "${MYFILE%/*}      =" "${myfile%/*}"
echo "dirname $MYFILE  =" $(dirname $MYFILE)
```

第 1 个 echo 语句删除文件名中匹配 “*/” 的最长字符串，从变量的头部开始，一直删除到最后的 “/”，仅返回文件名。第 3 个 echo 语句匹配所有的 “/” 之后的字符串，从变量尾部开始，仅删除文件名，返回文件名的路径。这个脚本的输出是：

```
$ ./pattern.sh
$ {MYFILE##*/}          = ide.txt
basename $MYFILE        = ide.txt
${MYFILE%/*}            = /usr/src/linux/Documentation
dirname $MYFILE         =/usr/src/linux/Documentation
```

为了说明模式匹配和替换操作符的用法，下面命令用空行来替换新行 \$PATH 环境变量中的每个冒号，以使得路径显示易于阅读（这个例子在低于 bash 2.0 或者更新的版本环境下不能运行）。

```
$ echo -e ${PATH//:/\\n}
/home/kwall/bin
/bin
/usr/bin
/usr/local/bin
/usr/X11R6/bin
/opt/bin
/opt/texTeX/bin
/opt/kde/bin
/usr/java/bin
```

当然，你的 \$PATH 变量的实际值可能看上去有点不一样，echo 里的 -e 参数告诉它将 \n 解释为新行，而不是普通的字符串。但是，和一般情况不太一样，为了让 echo 能解释，你必须对转义字符再进行转义 (\\n)，把新行的标识符插入变量中。

30.5 流 控 制

一些 shell 脚本至今只有顺序执行的脚本，缺乏诸如循环和条件在内的控制结构。任何有价值的编程语言必须有多次重复某个代码块的功能，以及根据条件执行某块代码操作或不执行某块代码操作的功能。在本节中，你将遇到所有的 bash 流控制结构，包括：

- if——如果条件为真或者为假，执行一个或者多个语句
- for——按固定次数执行一个或者多个语句
- while——如果条件为真或者为假，执行一个或者多个语句
- until——执行一个或者多个语句直到条件为真或者为假
- case——根据某个变量的值执行一个或者多个语句
- select——根据用户的选择执行一个或者多个语句

30.5.1 条件执行：if

bash 支持用 if 语句条件执行代码，虽然它计算条件的方法和 C 语言或者 Pascal 语言的 if 语句稍有不同。但是，除此之外，bash 的 if 语句和 C 语言里 if 语句功能一致。它的语法可总结如下：

```
if 条件
then
    语句
[elif 条件
    语句]
[else
    语句]
fi
```

首先，确定你理解 if 检查条件中最后语句的退出状态。如果是 0（真），那么语句将被执行，但是如果不是 0，而且有 else 语句的话，则执行 else 语句，之后控制将跳转到 fi 后面的第一行代码。elif 子句（可选）（你可以写任意多的 elif 语句）将仅在 if 条件为假时执行。同样的，所有 else 语句（可选）将在 else 语句失败时执行。一般情况下，Linux 程序在操作成功时返回 0，否则返回非 0，所以这种限制也不麻烦。

警告：并非所有的程序都遵循同样的返回值标准，因此如果你在 if 条件语句中测试返回代码，那么你需要查看程序的文档，弄清楚它的返回值标准。例如 diff 程序，没有不同时，它返回 0，有不同时，它返回 1，出错时返回 2。如果一个条件语句没能够按照预期来执行，那么应该检查退出代码。

不管程序怎样定义它们的退出代码，bash 认为 0 代表真或者正常，非 0 则与之相反。如果你特别需要检查和保存命令的退出代码，那么可以在运行一个命令后立即使用“\$?”操作符。“\$?”返回最近执行命令的退出代码。

由于 bash 允许在条件中使用“&&”和“||”操作符（可近似翻译为逻辑“与”和逻辑“或”）来组合退出代码，情况就变得更加复杂。假定在你输入一个代码块之前，你不得不

进入一个目录然后拷贝一个文件。一种方法就是使用 if 语句嵌套，例如下面的代码：

```
if cd /home/kwall/data
then
    if cp datafile datafile.bak
    then
        # more code here
    fi
fi
```

然而 bash 允许你把上面这段代码写得更简洁，就像下面这一小段代码描述的那样：

```
if cd /home/kwall/data && cp datafile datafile.bak
then
    # more code here
fi
```

上面两段代码完成了同样的工作，但是第二段代码短得多，而且我认为，第二段代码更清晰更容易理解。如果因为某种原因，cd 命令失败了，bash 将不会尝试 copy 操作。

if 条件语句仅能测试退出代码，但是你可以使用 [...] 结构或者 bash 内置的命令 test 来测试更为复杂的条件。[条件] 返回一个代码，表明条件为真还是为假。test 命令做同样的工作，但是我发现它更难读懂。

bash 自动提供的可用于测试的条件范围有 35 种，丰富而且完备，可用来测试各种各样的文件属性以及比较字符串和整数。

注意： [条件] 结构左括号后和右括号前的空格是必须的。这是 bash shell 语法的烦人要求。因此，有一些人继续使用 test。我的建议是学会使用 [条件]，因为这样你的 shell 代码的可读性将会更好。

表 30.4 列举了最常见的文件测试操作符（完整的列表可以从 bash 出色的手册页面上获得）。

表 30.4 bash 文件测试操作符

操作符	真值条件
-d file	file 存在并且是一个目录
-e file	file 存在
-f file	file 存在并且是普通文件（不是目录或者特殊文件）
-g file	file 存在并且是 SGID（设置组 ID）文件
-r file	对 file 有读权限
-s file	file 存在而且不为空
-u file	file 存在并且 SUID（设置用户 ID）文件
-w file	对 file 有写权限
-x file	对 file 有执行权限，如果 file 文件是目录，则有查找权限
-O file	拥有 file

(续表)

操作符	真值条件
-G file	测试是否是 file 所属组的一个成员
file1 -nt file2	file1 比 file2 新
file1 -ot file2	file1 比 file2 旧

程序清单 30.4 列出了一个脚本 descpath.sh, 这个 shell 脚本的示例使用文件测试操作符列举\$PATH 的每个目录下符合某些条件的目录。

程序清单 30.4 文件测试操作符

```
#!/bin/bash2
# descpath.sh - File test operators

IFS=:

for dir in $PATH;
do
    echo $dir
    if [ -w $dir ]; then
        echo -e "\tYou have write permission in $dir"
    else
        echo -e "\tYou don't have write permission in $dir"
    fi
    if [ -O $dir ]; then
        echo -e "\tYou own $dir"
    else
        echo -e "\tYou don't own $dir"
    fi
    if [ -G $dir ]; then
        echo -e "\tYou are a member of $dir's group"
    else
        echo -e "\tYou aren't a member of $dir's group"
    fi
done
```

上节讨论的 for 循环, 默认情况下使用空白(空格、新行和制表键)来分隔域或者标识符。设置了\$IFS (第6行), 即域分隔符为“:”之后, 将使 for 使用“:”分析它的域。对每个目录, 脚本测试你是否有写权限(第11行)、所有权(第16行), 以及是否组成员(第21行), 并且以好看的格式在每个目录名后打印这些信息(第12、14、17、19、22和24行)。在我的系统下, 该脚本的输出如下(当然, 在你的系统下, 输出会有所不同):

```
$ ./descpath.sh
/home/kwall/bin
You have write permission in /home/kwall/bin
You own /home/kwall/bin
You are a member of /home/kwall/bin's group
```

```

/bin
  You don't have write permission in /bin
  You don't own /bin
  You don't a member of /bin's group
/usr/bin
  You don't have write permission in /usr/bin
  You don't own /usr/bin
  You aren't a member of /usr/bin's group
/usr/local/bin
  You don't have write permission in /usr/local/bin
  You don't own /usr/local/bin
  You aren't a member of /usr/local/bin's group
/usr/X11R6/bin
  You don't have write permission in /usr/X11R6/bin
  You don't own /usr/X11R6/bin
  You aren't a member of /usr/X11R6/bin's group
/opt/bin
  You don't have write permission in /opt/bin
  You don't own /opt/bin
  You aren't a member of /opt/bin's group
/opt/teTeX/bin
  You don't have write permission in /opt/teTeX/bin
  You don't own /opt/teTeX/bin
  You aren't a member of /opt/teTeX/bin's group
/opt/kde/bin
  You don't have write permission in /opt/kde/bin
  You don't own /opt/kde/bin
  You aren't a member of /opt/kde/bin's group
/usr/java/bin
  You don't have write permission in /usr/java/bin
  You don't own /usr/java/bin
  You aren't a member of /usr/java/bin's group

```

bash 的字符串测试以字典顺序比较字符串。这意味着，例如，as 就要小于 asd。表 30.5 列举了字符串测试。

表 30.5 bash 字符串比较

测试	真值
str1=str2	str1 和 str2 匹配
str1!=str2	str1 和 str2 不匹配
str1<str2	str1 小于 str2
str1>str2	str1 大于 str2
-n str	str 的长度大于 0（不是空串）
-z str	str 的长度为 0（空串）

整数测试的功能和你所想到的一致。表 30.6 列举了这些测试。

表 30.6 bash 整数测试

测试	真值
val1 -eq val2	val 1 等于 val 2
val1 -ge val2	val 1 大于或等于 val 2
val1 -gt val2	val 1 大于 val 2
val1 -le val2	val 1 小于或等于 val 2
val1 -lt val2	val 1 小于 val 2
val1 -ne val2	val 1 不等于 val 2

30.5.2 确定性循环: for

正如程序清单 30.4 显示的, for 允许你以固定次数执行某段代码。但是 bash 的 for 控制结构只允许你使用固定长度的值列表对循环进行控制,因为它不能像 C、Pascal 或者 Basic 语言那样自动增加或者减少一个循环的计数器的值来进行循环控制。即使这样, for 循环仍然是经常使用的循环工具,因为它对列表的操作非常简单,例如命令行参数和目录下的文件。完整的语法如下:

```
for value in list
do
    statements using $value
done
```

list 是一个值的列表,例如文件名。value 是单个表项, statement 是 bash 使用或者操作 value 的表达式。

For 循环的一个典型的用法是在单个操作中重命名许多文件。有趣的是, Linux 缺少一个简便的方法来给组文件改名字。而在 MS-DOS 下,如果你有 17 个文件,每个都有*.doc 扩展名,你可以使用 COPY 命令来把每个*.doc 文件转换为*.txt 文件。DOS 命令如下:

```
C:\ copy *.doc *.txt
```

遗憾的是, Linux 的 cp 命令不能以相同的方式工作,因为 shell 解释特殊字符*的方式不同。使用 bash 的 for 循环来解决这个缺点。

下面的代码可以放入一个 shell 脚本中——起个有帮助意义的名字 copy——能精确地完成你想做的事件。

```
for docfile in doc
do
    cp $docfile ${docfile%.doc}.txt
done
```

使用 bash 的一个模式匹配操作符,这一小段代码将把以.doc 扩展名结尾的文件名转换为.txt 结尾的文件名。

30.5.3 不确定性循环: while 和 until

for 循环限制一个特定代码段的执行次数, 而 bash 的 while 和 until 结构当 (只要) 某个特定条件符合则执行代码。惟一的要求是你的代码或者程序运行的环境必须确保该条件最后能使循环安全退出, 否则会形成死循环。它们的语法如下:

```
while 条件
do
    语句
done
```

这个语法的意思是, 只要条件为真, 就执行语句直到条件为假 (直到程序或命令返回非 0 值):

```
until 条件
do
    语句
done
```

从某种意义上讲, until 语句的语法和 while 语句的语法正好相反: 直到条件为真, 才执行语句 (就是说, 直到命令或者程序返回非 0 值退出 do 和 done 之间的语句代码, 才去执行其他代码) 有经验的程序员会发现 until 循环会导致“忙等待”, 这对编程很不利, 而且经常浪费大量系统资源, 特别是 CPU 周期。另一方面, bash 的 while 结构为你提供了一种方法, 克服 for 结构不能自动增减计数器的弱点。例如, 假想你的刺儿头老板坚持某个文件要有 150 个备份。下面的例子片段使这种工作成为可能 (不用理会那个白痴老板)。

```
declare -i idx
idx=1
while [ $idx!=150 ]
do
    cp somefile somefile.$idx
    idx=$((idx+1))
done
```

除了能描述 while 的使用方法, 这段代码还介绍了 bash 的整数算法。declare 语句创建了一个变量 idx, 并将其定义为整数。每次循环迭代都将 idx 加 1, 保证循环条件最后为假, 程序控制能退出循环。这个脚本在光盘上存为 phb.sh。要删除它创建的文件, 执行 rm somefile*。

30.5.4 选择结构: case 和 select

下一个考查的流程控制结构是 case, 它和 C 语言里面的 switch 语句类似: 它允许你按一个变量的不同值来执行不同的代码块。完整的 case 语法如下:

```
case 表达式 in
    模式 1 )
        语句 ;;
    模式 2 )
```



```
        语句 ;;  
        .....  
    esac
```

表达式和每个模式比较，和第一个匹配上的模式关联的语句将被执行。“;;”号和C语言的break语句等价，使得程序的控制流跳转到esac后面的第一行代码执行。但是，和C语言的switch关键字不同，bash的case语句允许对表达式和含有通配符表达式模式进行匹配。程序清单30.5描述了case是如何工作的。

程序清单 30.5 使用case语句

```
#!/bin/bash2  
# case.sh - Using the case selection structure  
  
function usage  
{  
    echo 'Press a, b, c, or q. Type "q" and press ENTER to exit'  
}  
  
clear  
usage  
read OPT  
while [ true ]  
do  
    case $OPT in  
        a) echo 'You pressed "a"' ;;  
        b) echo 'You pressed "b"' ;;  
        c) echo 'You pressed "c"' ;;  
        q) exit 0 ;;  
        ?) clear && usage ;;  
    esac  
    read OPT  
done
```

case.sh 打印出一段简短的用法消息，然后进入等待输入的无限循环中。case 语句比较\$OPT 的值，打印输出一条相应的消息。示范运行的结果如下：

```
$ ./case.sh  
Press a, b, c, or q. Type "q" and press ENTER to exit  
a  
You pressed "a"  
b  
You pressed "b"  
c  
You pressed "c"  
d  
Press a, b, c, or q. Type "q" and press ENTER to exit  
q
```

`select` 控制结构（在 `bash` 1.14 以前版本中没有）只有 `korn` 和 `bash shell` 才有。另外，在常规的编程语句中没有它的类似语句。`select` 允许你构造一个简单的菜单，并且轻松地响应用户的选择。它的语法如下：

```
select value [in list]
do
    statements
done
```

程序清单 30.6 描述了 `select` 是如何工作的。

程序清单 30.6 用 `select` 创建菜单

```
#!/bin/bash2
# menu.sh - Creating simple menus with select

IFS=:
PS3="choice? "

# clear the screen
clear

select dir in $PATH
do
    if [ $dir ]; then
        cnt=$(ls -Al $dir | wc -l)
        echo "$cnt files in $dir"
    else
        echo "Dohhh! No such choice!"
    fi
    echo -e "\nPress ENTER to continue,CTRL-C to quit"
    read
    clear
done
```

这个脚本首先设置 `$IFS` 字符为一个冒号 (`:`)，以便 `select` 可以正确地分析环境变量 `$PATH`。随后，脚本将改变 `select` 使用的默认提示符，即内建的 shell 变量 `$PS3`，变为比“`#?`”更有帮助的值。在清屏之后，它进入了一个循环，显示一个由 `$PATH` 环境变量产生出来的目录菜单，并提示用户进行选择，如图 30.1 所示。

如果用户做出了一个有效选择，命令替换 (`cnt=$(ls -Al $dir | wc -l)`) 把 `ls` 命令的输出送入管道作为单词计算命令 `wc` 的输入，以计算出目录下文件的个数并显示结果。因为 `ls` 命令可以不带参数，所以脚本首先确定 `$DIR` 不为空。（如果它为空，`ls` 将对当前目录进行操作，即使用户作出了一个无效选择）。如果用户做出了一个无效选择，`menu.sh` 显示一条错误消息，随后提示说明怎样继续执行。`read` 语句将在后面 30.7 “输入与输出”一节中介绍，它允许用户查看输出并且耐心等待用户按下回车键来重复进行下一次循环，或者按下 `Ctrl+C` 键退出程序。

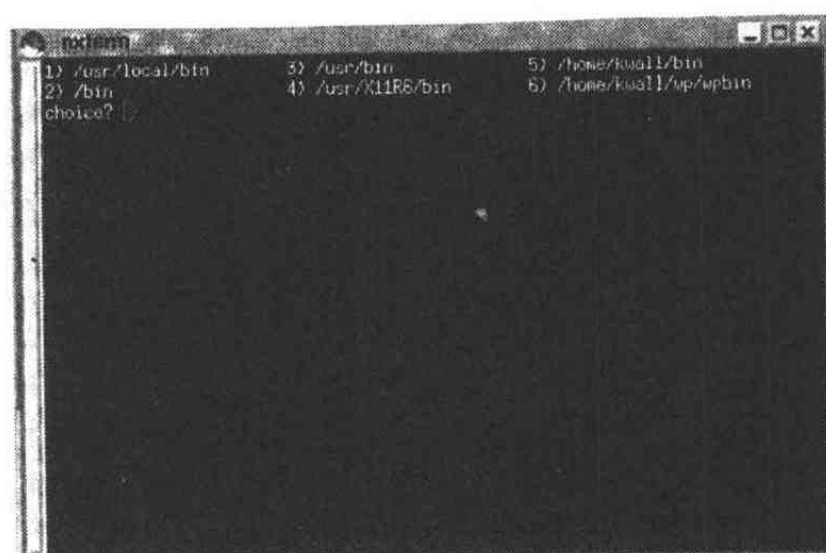


图 30.1 select 是创建简单菜单的一种方法

注意： 以上脚本循环将不会停止，除非按 Ctrl+C 键。但是，在用户输入合法的选项后使用 break 语句是一个正确的方法。

得益于编程语言，bash 已经拥有一个完备而且功能强大的流程控制结构。虽然它们和传统编程语言里的流程控制结构的作用稍有不同，它们还是给 shell 脚本增加了相当强大的功能和灵活性。

30.6 shell 函数

bash 的函数特性是其他 shell 函数功能的一个扩充版本。它有两大主要优势：

- 执行速度更快，因为 shell 函数已经装入内存。
- 模块化，因为函数帮助你总结归纳 shell 脚本，并且允许你更好地组织你的脚本。

可以使用下面两种格式的一种定义 shell 函数：

```
function fname
{
    commands
}
```

或者

```
fname ()
{
    commands
}
```

两种格式都是可行的而且两者没有功能方面的差异。通常的做法是在使用它们之前，在脚本开头定义你所有的函数，正如我在程序清单 30.2 里所描述的一样。要调用一个已经定义好的函数，简单地调用函数名称，后面加上所需参数就可以了。

要明确的是，和 C 语言或 Pascal 语言相比，bash 的函数接口非常原始。既没有错误检查也没有办法传递值参。但是，就像在 C 或 Pascal 里一样，在一个函数内部声明的变量可以作为函数的本地变量，从而可以用前向定义或者在第一次使用变量时用关键字 `local` 来避免名字空间冲突。如下面程序段所示：

```
function foo
{
    local myvar
    local yourvar=1
}
```

因为程序清单 30.2 包括一个 shell 函数声明和使用的例子，推荐你回去看看那个脚本。你可以做个实验，试着改变声明格式，说服你自己两种声明格式是等价的。

提示： shell 函数使你的脚本变得更容易阅读和维护。使用函数和注释，你可以在不得不回头改进那些你 6 个月前写的代码时免于痛苦不堪的阅读和维护工作。

30.7 输入与输出

限于篇幅，本节不会讨论所有的输入和输出操作符和功能。很多这样的操作符与功能和文件描述符、I/O 设备文件以及标准 I/O 的操纵有关。最常用的是 I/O 重定向操作和字符串 I/O 操作。

30.7.1 I/O 重定向

你已经看到了基本的 I/O 重定向操作符，“>”和“<”，它们分别重定向输出和输入。输出重定向允许你将一个命令的输出传送给一个文件。例如命令：

```
$ cat $HOME/.bash_profile > out
```

这个命令通过重定向 `cat` 命令的输出，在当前工作目录下创建一个名为 `out` 的文件，使该文件包含 `bash` 初始化文件 `.bash_profile` 的内容。

同样，可以用输入重定向操作符“<”，使得一个命令的输入来自一个文件或者命令。你可以用输入重定向重写前面的 `cat` 命令，如下所示：

```
$ cat < $HOME/.bash_profile > out
```

该命令的输出和前面的相同，而且说明你可以同时进行输入和输出重定向。

输出重定向操作符“>”，将改写任何存在的文件。有时候这不是你所想要的，所以 `bash` 提供追加操作符“>>”，将重定向数据加在文件尾部。下面命令将别名 `cdlpu` 加在 `.bashrc` 初始化文件的后面：

```
$ echo "alias cdlpu='cd $HOME/kwall/project/lpu'" >>$HOME/.bashrc
$ echo "alias cdlpu ='cd $HOME/lpu2'">> $HOME/.bashrc
```

here 文档是输入重定向的一个有趣的例子。它们迫使一个命令的输入为 shell 的标准输入。一个 here 文档是给一个 shell 脚本提供输入而不需要交互输入的方法。所有的读入行都成为脚本的输入。语法如下：

```
command<<label
    input...
label
```

这个语法的意思是命令 `command` 当遇到 `label` 作为独立的一行时读 `input`。here 文档用于当脚本程序接口不便于使用时进行脚本编程。FTP 程序是一个很好的例子。程序清单 30.7 描述了如何使用 here 文档编写 FTP 脚本程序。

程序清单 30.7 使用一个 here 文档进行 FTP 脚本编程

```
#!/bin/bash2
# ftp.sh - Use a here-document to script ftp

USER=anonymous
PASS=kwall@kurtwerks.com

ftp -i -n << END
open ftp.caldera.com
user $USER $PASS
cd /pub
ls
close
END
```

程序清单中的 `ftp` 用非交互模式 (`-i`) 启动 FTP，并且关闭自动登录 (`-n`)。脚本用 `END` 作为标志来标明输入结束。使用不同的 `ftp` 命令，脚本先打开一个 FTP 会话通往 Caldera System 公司的 FTP 网址。然后，它使用事先定义的 `$USER` 和 `$PASS` 变量发送登录序列。一旦登录成功，它就将目录改变为标准的公共 FTP 目录 `pub`，然后执行 `ls` 命令证实脚本工作完成。最后它关闭会话。直到遇到仅含 `END` 标志的行，脚本关闭向 FTP 的输入，同时退出 FTP 程序和脚本程序。还可以使用 here 文档，在脚本中包含文件。

30.7.2 字符串 I/O

你已经遇到了字符串输出中的 `echo` 语句。除了它用于启动转义字符如 “\n”（新行）和 “\t”（跳槽）解释的 “-e” 选项，`echo` 还可以接受 “-n” 选项来取消加在它输出后的新行。其他 `echo` 能够理解的转义序列都包含在表 30.7 里面。

表 30.7 each 转义序列

序列	描述
\a	Alert/Ctrl+G (bell)
\b	Backspace/Ctrl+H
\c	忽略附加到输出后的新行

(续表)

序列	描述
\f	Formfeed/Ctrl+J
\r	Return/Ctrl+M
\v	Vertical tab
\n	八进制值的 ASCII 字符, n 为 1~3 数字
\\	单个\

要处理字符串输入, 需要 `read` 操作符。它的语法是:

```
read var1 var2 ...
```

虽然理想的情况是用户以交互模式输入, 但是 `read` 也可以被用来一次处理文本文件中的一行。在交互模式下, `read` 的用法简单, 如下面的小段代码所示:

```
echo -n 'Name: '
read name
echo "Name is $name"
```

使用 `read` 来处理文本文件多少有些复杂。最简单的方法是创建一个脚本, 然后重定向你想处理的文件为它的输入。程序清单 30.8 处理 `/etc/passwd` 的内容。

程序清单 30.8 showpass.sh

```
#!/bin/bash
# showpass.sh - Using read to process a text file
#####

IFS=:

while read name pass uid gid gecos home shell
do
    echo "*****"
    echo "name   : $name"
    echo "pass   : $pass"
    echo "uid     : $uid"
    echo "gid     : $gid"
    echo "gecos   : $gecos"
    echo "home    : $home"
    echo "shell   : $shell"
done
```

设置 `IFS` 使得脚本能够正确解析 `/etc/passwd` 目录。while 语句读输入, 按顺序将各个域赋值给每个变量 (如果输入域的数量超过了接受它们的变量数, 最后的域都追加到变量表里最后的变量, 如此例中的变量 `shell`)。可以用至少下面两个命令中的一个来执行这个脚本:

```
$ ./showpass.sh < /etc/passwd
```

或者

```
$ cat /etc/passwd | ./showpass.sh
```

无论用哪个命令，输出将会是相同的。一个简略的输出形式如图 30.2 所示，管道使用了 more 命令来控制翻页。

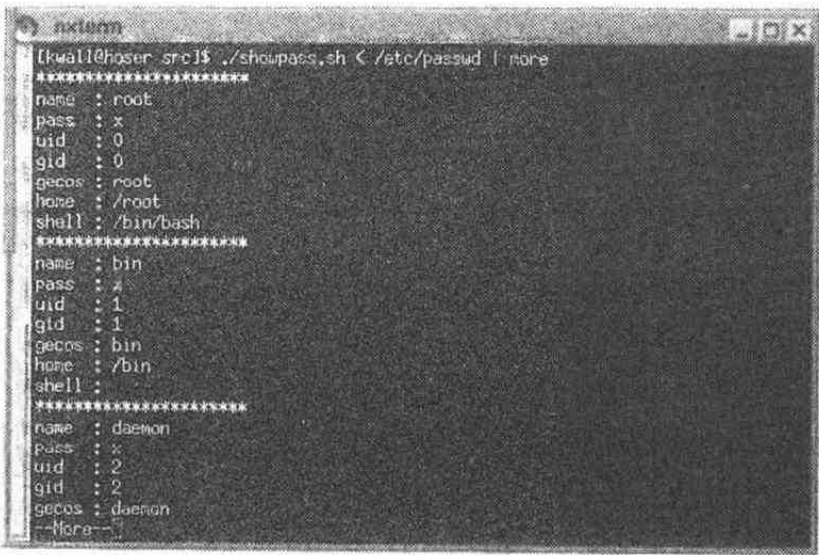


图 30.2 使用 read 语句处理文本文件

在高于 2.0 的 bash 版本中，read 接受一些选项来加强它的功能，这些选项列在表 30.8 中。

表 30.8 read 选项

选项	描述
-a	将值读入数组，数组下标从 0 开始
-e	使用 GNU 的 readline 库进行读入操作，允许使用 bash 的编辑功能
-p 提示	在执行读入前打印提示

使用 read 的 -p 选项，可以不用像下面的代码那样：

```
echo "Enter Name:"
read name
```

可以直接写成

```
read -p "Enter Name: " name
```

30.8 命令行处理

一个完善的 shell 脚本应该能够处理形如 -option 的命令行选项，以便模拟标准 UNIX 和 Linux 的命令格式。一个好消息是 bash 有一个内置的命令 getopt，使得处理任意命令行选项变得非常容易。有经验的 C 程序员将会意识到 bash 的 getopt 和 C 标准库的例程 getopt 非常相似。

getopt 有两个参数，一个由字母和冒号组成的字符串以及一个变量名。第一个参数是合法选项的列表；如果选项需要一个参数，那么参数后面必须跟一个冒号。getopt 分解第

一个参数，将选项摘取出来，然后依次将每个选项（没有选项前的划线“-”）赋值给第二个参数，第二个参数的变量名由用户赋予。只要选项仍在处理，`getopts` 就返回 0，但是当它处理完所有选项和参数后它返回 1，这使得 `getopts` 很适合配合 `while` 循环来处理命令行参数。程序清单 30.9 对此进行了更加具体的讨论。

程序清单 30.9 `getopts.sh`

```
#!/bin/bash
# getopts.sh - Using getopts
#####

while getopts ":xy:z:" opt;
do
    case $opt in
        X ) xopt='-x set' ;;
        Y ) yopt = "-y set and called with $OPTARG" ;;
        z ) zopt = "-z set and called with $OPTARG" ;;
        \?) echo 'USAGE: getopts.sh [-x] [-y arg] [-z arg] file ...'
            exit 1
        esac
    done
    shift ${OPTARG - 1}

    echo ${xopt: -'did not use -x'}
    echo ${yopt: -'did not use -y'}
    echo ${zopt: -'did not use -z'}

    echo "Remaining command-line arguments are:"
    for f in "$@"
    do
        echo -e "\t$f"
    done
```

正如所看到的那样，`getopts` 调用的有效参数为 `x`、`y` 和 `z`，而 `y` 和 `z` 后面必须跟参数。当 `getopts` 逐个摘出选项时，它把它们存入变量 `$OPT` 中，它作为 `case` 语句的测试值。`case` 语句处理每个有效的选项，包括默认的情况（?），此时如果使用了无效选项，则打印一条用法信息然后退出脚本。用 `y` 和 `z` 选项传递的参数保存在内置变量 `$OPTARG` 中。这些参数应该尽快保存到其他变量中，因为随着 `getopts` 顺序读取选项列表时，`$OPTARG` 的值也不断变化。脚本最后的 `echo` 语句只打印某个选项是否使用，以及如果有参数的话，还打印它的参数值。

第 16 行和第 22~25 行更多地展示了 `getopts` 是如何工作的。`$OPTARG` 是一个数，它等于下一个将被处理的命令行参数的位置。`shift` 内置命令移走位置参数，就是说，`shift N` 移走 `N` 个位置参数。因此，`$OPTARG-1` 和位置参数里选项参数的数目相等。所有的选项参数被移走，剩下非选项参数留给普通脚本处理。第 22~25 行打印剩下的位置参数以表明它们没有被动过。如果程序清单里的脚本在本章的源代码目录下被执行，该脚本的输出如图 30.3 所示。

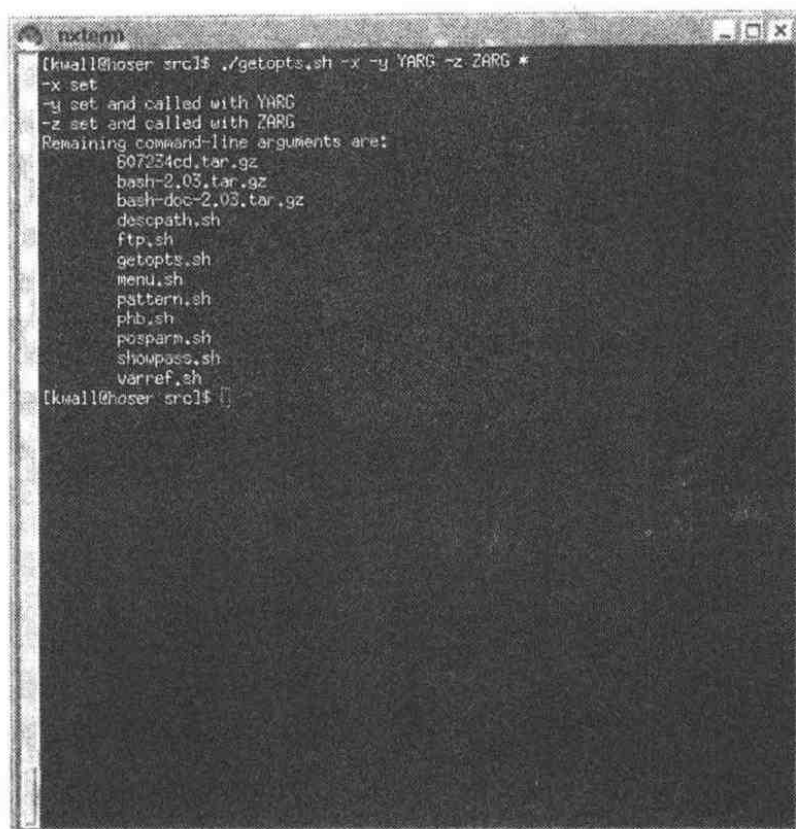


图 30.3 getopts 简化了命令行选项的处理

由于有了 `getopts` 语句使得在 shell 脚本里处理长而复杂的命令行变得轻而易举。结果，要使 shell 脚本和其他 UNIX 程序的行为相似（至少在调用约定上相似），也变得轻而易举了。

30.9 进程和作业控制

这一节继续集中讨论如何改善 shell 脚本。这次，将讨论使用 shell 内置命令 `trap` 来使你的 shell 脚本更经得起打击。

30.9.1 Shell 的信号处理

使用 `trap` 的主要动机是为了正确处理不正常事件，例如中断（Ctrl+C）和挂起（Ctrl+Z）。虽然你可能对写就仍掉的脚本编写细致的错误处理代码不感兴趣，但是即使是仅通知信号的简单代码也能使你的程序看起来更健壮。另一方面，如果这一章确实激起了你使用脚本编程的热情，你开始用 `bash` 编写低层的系统程序，那么 `trap` 将是你代码中的重要组成部分。（是的，你可以用 shell 代码完成系统编程。）

在 Linux 下，`bash` 可以识别大约 30 种信号。要看完整的信号清单，使用命令 `kill -l`。实际上，可能只需要对大概 12 种信号采取行动。在这一节中，将侧重于讨论 `SIGTERM`、`SIGINT` 以及 `SIGKILL`。

30.9.2 使用 `trap`

`trap` 命令的语法如下：

```
trap command sig1 sig2 ...
```

当信号 sig1、sig2 中的任意一个被接收到时，trap 就执行命令 command，command 可以是二进制文件、shell 脚本或者 shell 函数。当 Command 完成后，脚本重新执行。信号可以通过名称或者数字来标识。考查程序清单 30.10。它建立了对两个信号的处理，一个对 SIGINT，一个对 SIGTERM。

程序清单 30.10 简单的 trap 用法

```
#!/bin/bash

# trap.sh-Using trap to handle signals
#####

trap "echo 'Do not interrupt me!'" INT
trap "echo 'Attempted murder is illegal!'" TERM

echo "$0's PID is $$"
write true;
do
    sleep 30
done
```

除了描述 trap 的用法和行为，还加进了 bash 中和作业控制有关的特性：内置变量 \$\$，它描述当前执行脚本的进程 ID 号（PID）。如果脚本接收到有关 SIGINT（中断或 Ctrl+C）信号，第 1 条 trap 语句建立执行代码。第 2 条 trap 语句为 SIGTERM 处理函数建立代码，这个信号不能普通地从键盘产生，而必须使用一个命令，比如 kill \$(pidof trap.sh) 来产生。while 循环是一个死循环。因为 true 是 bash 的内置命令，总返回 0；每隔 30 秒，循环重复它自己一次。打印 shell 脚本的进程 ID 号，所以知道哪个进程 ID 使用了 kill 命令。

如果在前台运行该脚本，然后尝试用 Ctrl+C 键中止它，输出如下：

```
$ ./trap.sh
./trap.sh's PID is 25655
[type CTRL-C]
Do not interrupt me!
```

你运行的时候显示的 PID 当然会有不同。现在，把脚本放到后台，然后尝试用 kill 命令的默认信号 SIGTERM 杀掉它：

```
[type CTRL-Z]

[1]+ Stopped          ./trap.sh
$ bg
[1]+  ./trap.sh &
$ kill %1
Attempted murder is illegal!

$kill -KILL %1
[1]+ Killed          ./trap.sh
```

如上所示，你收到了预想的输出。因为脚本没有为 SIGKILL 信号安装处理函数，可以用 kill -KILL 语法杀死它。

为了决定代码中要处理哪些信号，需要看看手册页面的第7部分 (man 7 signal)，它详细描述了 Linux 支持的每种信号。通过看手册页面，可以使你的 shell 程序健壮而完备。

30.10 小 结

本章讨论许多低层的内容，介绍了在 bash 下进行 shell 编程的基础知识。在介绍了 shell 的基本概念，比如通配符、特殊字符以及花括号扩展之后，讨论了变量和位置参数的使用。关于运算符一节展示了如何使用 bash 的字符串匹配和模式匹配运算。流控制结构，比如 if、while 和 case 能让你创建更为灵活、粒度更好的 shell 脚本，shell 函数也能起到同样的作用。还学到了在 shell 代码中如何读取输入和产生输出。最后，学习了两种能够让你的脚本程序显得很专业的方法。getopts 能让你轻松地处理复杂的命令行，而 trap 能让你设定信号处理函数。要对 bash 编程做进一步的了解，可参考 bash 的手册页面，以及随源代码一起发布的扩充文档。

第 31 章 设备驱动程序

本章讲述如何编写一个内核级的设备驱动程序，明确地说是编写以可加载内核模块的形式出现的设备驱动程序。将给出一个实际的步进电机控制器的设备驱动程序。

编写设备驱动程序不但要求你具备良好的编程技术，而且要求你对要打交道的硬件设备有充分的认识。不能理解底层设备的所有特殊之处将会导致各种问题——从系统变慢到硬件的彻底故障。我已经有一个硬件可以用于本章讨论的例子，在对它很熟悉也知道它设计良好的基础，我决定构造自己的硬件设备驱动程序。

对这个驱动程序可能有三个层次的试验。你可以根据包含的电路原理图（见本章后面的图 31.1）做出实际电路。你可以在一个没有连接实际电路的空闲的打印机并口上运行这个驱动程序，并可选择使用逻辑探针、示波器或类似的设备查看并口的信号。或者，可以在禁止端口 I/O 操作的情况下运行这个驱动程序，这能让你在没有空闲打印机并口的条件下试验这个驱动程序。

之所以选中步进电机，一方面是因为我个人需要使用这样一个驱动程序，另外也因为这是一个我所能想到的，能够说明实现一个真正的设备驱动程序所需要的大多数编程技术的最简单的驱动程序；许多其他的驱动程序可能不是过于简单，就是过于复杂。选中它也是因为我可以同驱动程序一起提供关于硬件的文档。

31.1 驱动程序的类型

对于许多设备来说，并不一定需要内核级的设备驱动程序。但是，在需要它们的情况下，就会有多种不同的类型。本节介绍不同类型的设备驱动程序，以及它们如何同内核一起使用。

31.1.1 静态链接的内核设备驱动程序

基本上有两种类型的底层内核驱动程序，其中的第一种是静态链接的类型。这种设备驱动程序直接编译和链接到内核中。静态链接的模块，一旦编译进入了内核，就始终附加在内核上，直到重新编译内核为止。这主要用于这样的设备驱动程序：没有它们提供的功能，你的系统就不能运行——例如，你的根分区所在硬盘的驱动程序。

可加载内核模块（Loadable Kernel Modules, LKM）能够被加载和卸载而不必重新链接内核，而且最重要的是，不需要重新启动你的计算机。这就能让你动态地配置系统。编写一个静态链接的驱动程序和编写一个可加载的内核模块几乎相同。静态链接和内核模块在本质上非常相似，但是 LKM 是目前更可取的方法。

31.1.2 可加载的内核模块

可加载内核模块能够在系统正在运行的同时被加载和卸载。这一重大改进克服了这样的缺点，在每次你要测试一个驱动程序的新版本时，都要在修改、重新编译并安装一个新内核后重启系统。因为它是可选的，并且在运行时加载，一个 LKM 在不被使用时不会使用内核内存，能够轻易地离开内核而独立发布。但是，如果选择只以二进制模块形式发布你的模块，就不得不为内核的每一种版本重新编译和发布一份模块。

31.1.3 共享库

在有些情况下，驱动程序可以作为一个共享库来实现，但如果驱动程序需要特殊的权限或者有特殊的时限需要则不行。这种共享库一般可供那些使用标准低层驱动程序（如通用的 SCSI 驱动程序）和硬件通信的高层驱动程序使用。

在第2章里提到过的 SANE 扫描仪库就是一个基于共享库的驱动程序系统的例子。这个通用 SANE 扫描仪共享库选择适当的专用模型共享库并且动态地加载它。这个专用模型共享库利用通用 SCSI 内核驱动程序和扫描仪通过 SCSI 总线通信。

如果想为一个扫描仪或者类似的成像设备编写驱动程序的话，就写一个 SANE 驱动程序。调试时你会发现，把程序构造成一个简单的、无特权用户模式的程序，然后再添加 SANE 界面，会更容易一些。

31.1.4 无特权用户模式程序

程序代码在内核模式或用户模式下执行。前面的那些类型运行在内核模式，而其他的类型运行在用户模式（或称用户空间）。运行在内核模式的代码对硬件有无限的低层访问权，而对高层（如文件和 TCP 网络连接）的访问就不那么容易实现。

许多设备通过像 SCSI、IDE、串口、并口（如果它们使用标准并口协议的话——许多设备不是这样）、USB、IRDA 等这样的标准通信通道连接到计算机上。这些设备的高层驱动程序通常不需要任何特权，除了写访问相应的/dev 文件之外，而这种访问通常放松了对特权的要求。

打印机驱动程序是这类驱动程序的一个好例子。为了给一个打印机编写驱动程序，你要给 Ghostscript PostScript 解释器写一个驱动程序模块。也可以用一个以 PBM (Portable BitMap) 文件（或称为 stream）为输入的独立 (standalone) 程序实现它。Ghostscript 可以被配置成以 PBM 文件输出，于是你可以修改打印队列以把 Ghostscript 的输出输送 (pipe) 到你的独立 (standalone) 程序。

大多数 RS-232 设备属于这一类。内核 RS-232 驱动程序提供对普通无智能串口以及智能多串口卡的低层接口。可以归于此类设备的例子有 PDAs、气象站、GPS 接收机、某些 voice/fax/data 调制解调器、某些 EPROM 编程器和串口打印机（包括标签打印机 (label printer)）。我的数码相机使用这种类型的驱动程序，尽管 SANE 驱动程序可能会更为合适。像第2章提到的 X10 和 Slink-e 设备也可归于此类，它们是控制电灯、咖啡壶、CD 转换器、VCR、接收机及其他家用或办公室用器具的 RS-232 设备。

31.1.5 特权用户模式程序

如果需要“原端口 I/O”这样的特权，那么可以使用一个以 root 身份运行的用户模式程序，特别在早期试验阶段。如果你使这个程序 suid，或相反地允许一般的用户（或者更糟糕的是允许一个远程用户）控制或和这个程序通信，那就会有严重的安全性问题。

31.1.6 守护进程

特权或者无特权的用户模式驱动程序可以扩展为守护进程。在一些情况下，守护进程可完全独立地运行。在另外一些情况下，你会允许本地或者远程用户同守护进程通信；在这种情况下，则需要提高警惕，以防止出现安全问题。要了解有关守护进程的更多信息，请参考第 18 章。

31.1.7 字符设备与块设备的对比

大多数 Linux 内核驱动程序或许要实现字符特殊设备。应用程序通过向用 mknod 命令在/dev 目录下创建的字符特殊文件读写数据流与这些设备通信。

块模式驱动程序用于磁盘和 CD-ROM I/O 及类似的操作。它们一般用于那些你可能在其上安装 (mount) 一个文件系统的设备。本章将不包括块模式驱动程序；在大多数情况下，支持新设备将只需要对现存的驱动程序进行修改。

SCSI 主机适配器的驱动程序既要实现块设备（用于磁盘 I/O）又要实现字符设备（用于通用设备接口）。它们通过现存的 SCSI 基础结构达到这些目的。网络接口驱动程序又是另一种不使用一般的字符或块模式接口的类型。声卡驱动程序使用字符设备接口。PCI 总线、PCMCIA 卡和 SBUS（用在 Sparc 系统上）的驱动程序有它们自己的特别接口，它们还为其他驱动程序提供基础结构。

设备驱动程序还可以写成实际不和任何硬件设备交互的形式。null 设备/dev/null 就是一个简单的例子。设备驱动程序可以用来加入通用的内核代码。一个设备驱动程序可以加入新的系统调用或替换原有的系统调用。查看文档/usr/src/linux/Document/devices.txt，了解包含在 Linux 内核中最常用的设备驱动程序的清单。

31.2 怎样构造硬件

为了演示如何编写一个设备驱动程序，我将使用非常简单的小硬件：一个三轴步进电机的驱动程序。步进电机是一类接受简单的计算机控制的电机。这个名字源于这些电机不连续步进的事实。通过给不同的线圈组合供电，计算机可以命令电机按顺时针方向或逆时针方向前进一步。

步进电机用于各种计算机外围设备中。表 31.1 显示了步进电机通常是如何被使用的。

表 31.1 步进电机在计算机外围设备中的功能

外围设备	功能
打印机	进纸, (固定纸的) 托架移动
绘图仪	笔和纸在 X 和 Y 轴上的移动
扫描仪	移动扫描仪滑架, 过滤轮 (3 步扫描仪)
某些磁带驱动器	相对磁带移动磁头
软盘驱动器	磁头定位
老的硬盘驱动器	磁头定位

许多计算机外围设备, 都有一个用来控制步进电机的板上处理器, 于是在这种情况下你不需要直接驱动电机。其他设备没有板上智能 (没有微处理器); 这种情况下你将需要直接控制电机, 于是就需要为设备编写一个驱动程序。

步进电机也被广泛应用于“机器人”和工业系统中。我编写这个驱动程序用来控制电脑化立式铣床。这个机床可以用来按要求把金属、塑料、电路板和材料切割成各种形状。X 轴和 Y 轴移动工件, Z 轴移动旋转的切刀刀头上下移动。

31.2.1 理解步进电机的工作原理

为了编写一个设备驱动程序, 你需要尽可能多地了解硬件是怎样工作的。我们需要深入考察步进电机。在本节中, 你将学习步进电机内部的工作原理。

一个简单的步进电机有两个互成 90° 的电磁线圈。每个线圈可以通断电并且可以有两种极性。为了简化驱动电路, 尽管牺牲了一些性能, 大多数步进电机使用中心抽头线圈。通过把中心抽头连接到正极, 而把线圈的一端或另一端接地, 你可以有效地使线圈按正负极性磁化 (于是反转了磁场的极性)。这些类型的电机被称为单极电机。单极步进电机有 5、6 或 8 条导线。其中四条导线将直接连到驱动晶体管。剩余的所有导线将连接到电源正极。

你可以用几块铁、一些导线和一个指南针制作一个粗糙的步进电机; 实际上, 你还可以省去那些铁。如果你尝试这种方法, 一定要使用一个限流电阻, 以防止烧坏磁场线圈、损坏驱动电路或电源, 或以不同的指向重新磁化指南针。设想你拿着那个指南针, 旋转的轴向着你。现在绕着指南针的上下边缘绕铜线做一个线圈, 绕左右侧做第二个线圈。

图 31.1 显示了我的没有修饰的步进电机驱动器电路。这个电路也可以用来驱动继电器、电灯、螺线管和其他设备。更复杂的电路会提供更好的性能 (更大的扭力和更快的速度) 和安全性, 但是这种每个电机只有四个晶体管和四个电阻的电路便宜、器件数少, 而且容易构造和理解。这个电路可以用不到 25 美元的器件做成。

不管你是否想编写一个步进电机的驱动程序, 这个电路都提供了一个简单设备, 它可以用来说明如何开发 Linux 设备驱动程序。

原理图上有一个插图, 显示了一个电机经由 8 个半步被驱动。B 和 B* 相位被画成相反的 (因此我不需要画出导线的绕法), 于是驱动程序实际上可以按照插图的示例使电机沿相反的方向旋转。你可以在驱动程序中通过改变步进表改变电机的旋转方向。

主要有三种方法驱动一个四相 (双线) 绕线步进电机。你可以想像四个 (半个) 线圈向北、东、南、西四个方向 (不要和南北磁极相混淆) 拉动电机的转子。单相位驱动按照

北 (A)、东 (B)、南 (A*)、西 (B*) 的顺序每次给一个线圈供电；括弧中的相位名和原理图上的相对应。双相位驱动使用更多的电能，同时通过用东北 (A+B)、东南 (A*+B)、西南 (A*+B*)、西北 (A+B*) 的顺序提供更大的扭力和更快的操作。半步步进用北 (A)、东北 (A+B)、东 (B)、东南 (A*+B)、南 (A*)、西南 (A*+B*)、西 (B*)、西北 (A+B*) 的顺序提供大的扭力、高速度和两倍的步进精度。半步步进将被用于这个驱动程序。当前面所说的顺序结束时，你只要按同样的顺序重复。要想反转旋转的方向，只需把顺序反转。要想让电机保持静止，只需在期望的位置暂停上面的顺序，并继续给线圈供电。

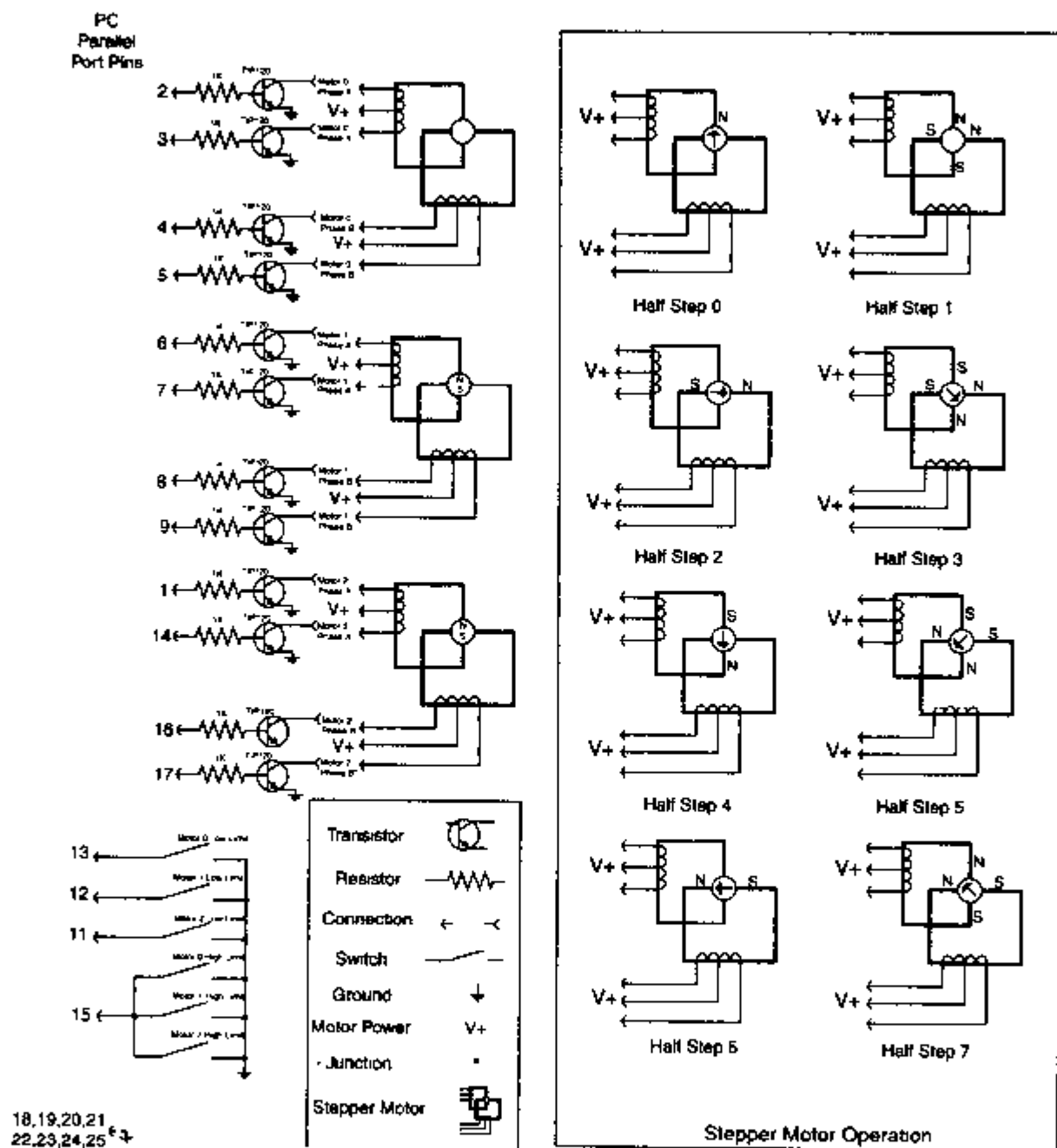


图 31.1 步进电机驱动程序电路

前面描述的示例电机每个机械步进精度都有一个电步进精度。大多数的实际电机每个机械步进精度有多个电步进精度，这通过损失速度而增加了扭力和步进精度。大多数通常的电机每英寸有 24 步（48 个半步）或 200 步（400 个半步）。当我做我的铣床时，我把每步进精度 400 个半步的电机和每英寸 20 转螺旋推动结合使用；这使每英寸移动 8000 步。

警告： 使用此电路不当可能损坏电脑、电路、电机，甚至引起火灾。

请使用电压和电流额定功率和你的步进电机相匹配的电源。这个驱动器电路没有防止同时给所有四个线圈同时供电的保护措施，这在有些时候会损害电机和电源，甚至可能由

于过热而导致火灾。可把电源电压减低到电机额定功率的 70%，通过牺牲部分性能来防止这种情况；有些电机的规格已经由于这个原因做了减免。确保电机线圈的电流汲取不会超过电流的额定功率和晶体管的电能耗散；给晶体管安上散热器。

步进电机在运转时会在线圈上产生电压峰。这个简单的电路没有限制这些电压峰，而是使用了一个能够忍受这些预期的电压峰的晶体管；检查一下以确保你使用的电机不会产生大到可以损害晶体管或电机的绝缘层的电压峰。可能需要用 TIP122 或其他有较高额定电压的合适的晶体管，或者添加突波抑制。如果你在电机运转时触摸电机连接，这些瞬时电压会产生一个疼痛的电击。

这个电路中使用的晶体管有很高的增益（高于 1000）；不要替换成低增益的晶体管。有些驱动能力较弱的并口也许不能提供足够的电流驱动晶体管，导致对并口或晶体管的损害。建议使用分离的、可被容易而且经济地替换的并口打印机卡以防造成损坏；膝上电脑上的内置并口换起来会特别昂贵。不要断开并口线，除非电脑和电机的电源都关掉了。接地不当可能会导致多个接地回路。电路应当由有资格的电子技师或者至少一个精通的业余爱好者组装。

多数情况下晶体管上需要散热器。我实际上把电路构造在一块四英寸厚的铝板上，并且用云母绝缘层把晶体管安装在铝板上，晶体管的安装（如果你不想电击击穿晶体管的话）及螺母、螺栓上使用了绝缘肩型垫片。然后，我把电阻直接焊接到晶体管的联接线上，把一些端接线条安在晶体管上的平衡器上以连接到电机。这个构造通过铝板提供了散热器，并且不需要面包板或印刷电路板。

控制端口上的四个输出线往往有 4.7K 的 5V 上拉电阻。你可能需要更小的 5V 上拉电阻为晶体管提供更大的驱动电流，而我不需要做这些。电机用的电源可以用实验室电源、一节大电池，或者，（对于小的电机）甚至可以用 PC 机的 12V 电源（在磁盘的电源连接线上可以得到）。

一个外时钟（脉冲产生器）可以连接到并口 Ack 线上，用来为步进电机提供定时。这为在快于每秒 100 脉冲（Linux 瞬时时钟的速度）的速度上运行电机提供了一个时钟源。Linux 实时扩展也可以用于这个目的。

31.2.2 标准的或双向的并口

这个电路使用向 PC 机的打印机并口的原始输出（raw output）。我们不能使用普通的打印机驱动程序，因为我们用非标准的方法使用这些信号。你不能把其他设备和这个电路菊花式链接到同一个并口上，所以不要试图在同一个并口上使用打印机、Zip 驱动器、CD-ROM，或其他设备。

我将描述 PC 机并口标准或双向模式的一般操作。在某些并口上可用的 EPP 或 ECP 模式的运行多少有些不同。你可能需要通过跳线设置或者机器中的 BIOS 设置把并口配置成标准或双向模式，以采用适当的操作。

双向模式允许 8 个数据线用作 8 个输入或 8 个输出，而不只是 8 个输出。控制寄存器中的 C5 位设定传输方向。有些用于提供并口的芯片使 C5 位无效，除非它们被用针对这个芯片的特殊的设置程序编程为双向模式；这样做是为了防止老的、拙劣的程序意外地改变并口的传输方向。

PC 机的并口通常被定位开始于 0x3BC (lp0)、0x378 (lp1) 或 0x278 (lp2) IO 端口地址。注意这些端口地址可能不直接对应在 DOS 和 Windows 中使用的 LPTx 编号, 这是由于当 lp0 不存在时它们把 lp1 重映射为 lp0, 并把 lp2 重映射为 lp1。0x3BC 地址传统地用于一些视频卡上的并口。对并口编程使用位于三个连续 I/O 端口地址的三个寄存器。

表 31.2 显示了在三个寄存器中每一位的名称。表 31.3 显示了每一位的功能。

表 31.2 并口编程接口

寄存器	地址	D7	D6	D5	D4	D3	D2	D1	D0	读	写
Data	base+0	D7	D6	D5	D4	D3	D2	D1	D0	Yes	Yes
Status	base+1	S7	S6	S5	S4	S3	S2	S1	S0	Yes	No
Control	base+2	C7	C6	C5	C4	C3	C2	C1	C0	Yes	Yes

表 31.3 并口硬件接口

位	信号	Pin
D0	Data bit 0	2
D1	Data bit 1	3
D2	Data bit 2	4
D3	Data bit 3	5
D4	Data bit 4	6
D5	Data bit 5	7
D6	Data bit 6	8
D7	Data bit 7	9
S0		
S1		
S2		
S3+	-Error	15
S4+	+Select	13
S5+	+PaperOut	12
S6+	-Ack	10
S7--	+Busy	11
C0--	--Strobe	1
C1--	--AutoFeed	14
C2+	--Init	16
C3--	+SelectIn	17
C4	IRQ Enable	none
C5	Output Enable	none
C6		
C7		
None	Ground	18~25

第一列的负号表明在计算机和并口之间有一个反向器。第二列的负号意味着这条线在对打印机接口时被认为低电平有效。

表 31.4 显示了步进电机到并口信号的连接。

表 31.4 步进电机驱动器连接

函数	姓名	位	Pin
Motor 0,Phase A	D0	D0	2
Motor 0,Phase A*	D1	D1	3
Motor 0,Phase B	D2	D2	4
Motor 0,Phase B*	D3	D3	5
Motor 1,Phase A	D4	D4	6
Motor 1,Phase A*	D5	D5	7
Motor 1,Phase B	D6	D6	8
Motor 1,Phase B*	D7	D7	9
Motor 2,Phase A	Strobe	C0*	1
Motor 2,Phase A*	AutoLF	C1*	14
Motor 2,Phase B	Init	C2	16
Motor 2,Phase B*	SelectIn	C3*	17
Motor 0,Low limit	Select	S4	13
Motor 1,Low limit	PaperEnd	S5	12
Motor 2,Low limit	Busy	S7*	11
All motors, High limit	Error*	S3	15
External Timer	Ack*	S6	10
Grounds			18~25

31.3 建立开发环境

为了开发内核设备驱动程序，建议有两台计算机分别运行相同版本的 Linux，并指定一种在它们之间传输文件的方法。在做驱动程序测试时很容易使一个系统崩溃，结果有可能造成对文件系统的损害。一个无限循环在一个用户模式程序中没什么大不了，但是在设备驱动程序的下半部，就可能使系统挂起。目标系统不需要很强大（除非你的驱动程序需要许多的 CPU 周期）；你可能有一个丢弃在一边的老的破系统，这就足够了。这个驱动程序是在一个有 4MB 内存的 386DX25 上测试的。虽然这对于运行驱动程序已经足够，但是，加载 emacs 花了 30 多秒，而编译花了近 4 分钟；内存是限制的因素。如果你没有在台机器上运行同样的内核版本，你需要在目标系统上重新编译。可以用软盘、网络连接或其他合适的方法把文件传输到目标系统。

提示： 你可以通过在加载驱动程序模块之前发出 sync 命令，减少崩溃事件中目标系统文件系统损坏的可能性。有时候，我甚至运行 while /bin/true; do sync; sleep 1; done & 来重复地 sync 文件系统。

31.4 调试内核级驱动程序

理论上可以用你的开发机器和目的机器之间的一个 RS-232 连接,以远程调试模式使用 GNU 的调试器 gdb。有关这个主题的更多信息可以在 gdb 的 info 页面中找到。

printk 函数和 printf 类似,但在控制台上显示输出。要谨防产生过多的输出,否则你可能会牢牢地锁上你的系统。建议你在代码中插入像 `if(debug>=5) printk(...)` 这样的语句。这将使你更容易指定模块加载时调试信息的级别。把这些编译进去是值得的,以使你的驱动程序的用户可以容易地调试问题。

提示: 如果你担心你程序中调试代码的额外开销,有一个简单的方法可把这些代码编译掉而不需要丢弃带有 `#ifdef DEBUG` 语句的代码。改为使用 `if(debug>=1)` 的形式。然后使用单独的一句 `#ifdef NODEBUG` 把 debug 的声明从 `int` 改为 `const int`。优化程序(如果允许的话,它对内核模块将总是使能的)应该知道这句代码永远不会被执行并把它从生成的可执行代码中删除。如果希望的话,这种设置允许在运行时设定调试级别,或者可以把调试部分全部编译掉。

你可以在模块用 `insmod` 加载时设定任何全局的或静态的变量。这可以通过在你调用的模块名后添加变量和它们的值来完成。例如:

```
insmod module.o debug=on
```

这使你不用重新编译就可以设定改变程序操作的各种调试变量。如果你试图在一段代码中定位崩溃,则可以用 `if(test1) {`, `if(test2) {`, 等等把各种小代码块括起来。然后你可以通过 `insmod` 这段代码并选择性地定义这些变量为 1 直到崩溃发生,即可快速定位问题所在。查看 `insmod` 手册页查看更多消息。

最后,我打算做一个符号表库(如在第 10 章所描述的)的可加载内核模块版本。这可以允许你当驱动程序运行时使用 `/proc` 文件系统交互地设置或查询不同的变量。它还提供了一种方法让用户配置驱动程序运行中的特性。两个独立的项可以在 `/proc` 中建立;一个只对 root 可访问并且可访问多个变量,另一个可以是任何人可访问并且可访问较少的变量。

31.5 设备驱动程序内幕

在实际编写一个设备驱动程序之前,需要介绍几个概念。这些概念将在本节予以讨论。首先,你需要知道怎样和硬件通过 I/O 端口进行通信,以及怎样使用 DMA 访问内存。然后需要知道怎样使用终端。最后,需要知道为什么设备驱动程序被分成了几层。

31.5.1 低层端口的 I/O

许多系统,包括 Intel 处理器,对内存和 I/O 端口有独立的地址空间。I/O 端口看起来有些像内存单元,但是它们可能不能读回和所写的值相同的值。这给粗糙的操作系统(例如 DOS)上的那些愿意做“读/修改/写”操作来改变一位或多位而不改变其他位的驱动程

序造成了一些问题，因为它们不知道其他程序把这些位设成了什么状态。在 Linux 下，对特定设备的 I/O 通常由一个单独的设备驱动程序完成，所以这一般不会成为问题。I/O 端口位置通常映射到系统上外围芯片的数据、控制或状态寄存器。有时候读或写某些 I/O 端口位置可能使系统挂起。如果你从某些 I/O 端口读，有些 NE2000 以太网适配器会使处理器永远挂起。这些卡被设计成插入等待状态直到请求的数据可用（如果你读一个未定义的寄存器它就永远不可用）。读写可能有副作用。读串口 UART 芯片上的数据寄存器将返回接收队列的下一个字符，同时有把这个数据从队列中删除的副作用。向相同的寄存器写则有把一个字符排队传送的副作用。

发出 `iopl(3)` 调用以便在以 `root` 身份运行的用户模式程序中使能低层 I/O。这给予了对 I/O 端口无限制的访问权。例如，如果你只需要访问 `0x3FF` 之下的端口，就可以使用 `ioperm(0x378,4,1)` 来打开端口 `0x378-0x37B`。第二种方法更有限制性，并将防止你意外地向不想写的端口写。

`outb` 函数向一个 I/O 端口输出一字节的值。函数调用 `outb(0x378,0x01)` 将把值 `0x01` 输出到端口 `0x378`。注意这两个参数的顺序和 DOS 程序员过去常用的正好相反。函数调用 `inb(0x378)` 将返回从端口 `0x378` 读出的一个字节。除了后面加了一个简短的延迟之外，函数 `outb_p` 和 `inb_p` 与 `outb` 和 `inb` 一样；许多外围芯片不能以总线全速处理端口读写。函数 `outw`、`outw_p`、`inw` 和 `inw_p` 是类似的，但适合于 16 位的值；`w` 表示 `word`。函数 `outl`、`outl_p`、`inl` 和 `inl_p` 也是类似的，但适合于 32 位的值；`l` 表示 `long`。I/O 端口操作的长度应当和设备支持的长度相匹配。你需要 `#include <asm/io.h>` 以便使用这些函数。

由于这些函数在 `gcc` 中实现的特性，使用这些输入和输出函数的程序，无论用户模式还是内核模式，必须带优化编译（对 `gcc` 来说，要用 `-O` 选项）。

函数 `insb(port,addr,count)` 和 `outsb(port,addr,count)` 用 Intel 兼容处理器上的有效率的重复串 I/O 指令在端口 `port` 和位于 `addr` 的内存之间移动 `count` 字节。这些函数没有间歇版本存在，这些指令的特点是在总线允许的情况下应尽可能快地移动数据。`word` 和 `long` 的版本也存在。使用 `long` 版本 `insl` 和 `outsl` 应当允许你在 33MHz 的 PCI 总线（其处理器将轮流进行 I/O 和内存操作）上以大约 66MB/s 的速率传递数据。

谨防从数据端口读取多于可用数据的数据；依靠所讨论的硬件，你得到一些垃圾或者设备将插入等待状态直到数据可用（可能在下个星期）。检查适当的状态寄存器里的标志以决定是否有一字节可用。如果你知道一定数量的字节可用，便可以使用快速的串来读取它们。

对内存映射的 I/O 设备有类似的指令：`readb`、`readw`、`readl`、`writeb`、`writew`、`writel`、`memset_io`、`memcpy_fromio` 和 `memcpy_toio`。在 Intel 兼容的处理器上这些函数实际上没有做任何特殊的事情，但它们的使用将使向其他处理器的移植变得容易。内存映射的 I/O 操作可能需要特殊的处理以确保它们没有被高速缓存所“碾压”。内存地址可能需要在总线地址、虚拟内存地址和物理内容地址之间，用函数 `bus_to_virt`、`virt_to_bus`、`virt_to_phys` 和 `phys_to_virt` 映射。

许多以太网设备需要为每一个包计算校验和。在现代的处理器上，这一般可以在以总线全速拷贝数据时完成。函数 `eth_io_copy_and_sum` 和 `eth_copy_and_sum` 被用于此目的。查看现存的以太网驱动程序，看一下如何使用这些函数。现代处理器经常受限于内存和 I/O

带宽；在移动数据时处理数据通常更有效率，这样使用了可能会被浪费的处理器周期。

31.5.2 使用 DMA 访问内存

直接内存访问（Direct Memory Access, DMA）允许内部的外围设备以不需要处理器为每一次传输执行指令的方式访问内存。主要有两种类型的 DMA：一种使用主板上的 DMA 控制器，另一种使用外围卡上的总线主设备控制器。

如果可能的话，建议你避免使用主板 DMA 控制器。只有有限数目的 DMA 通道，这可能使你难以找到空闲的通道。这种形式的 DMA 很慢，而且这种形式还要求短的中断延迟时间（在外围声称中断和处理器响应之间的时间）；当 DMA 控制器到达缓冲区的末端时，你需要快速地对它重新编程使其使用另外一个缓冲区。这个控制器每次传输需要多个总线周期：它从外围设备分别以两次独立的操作读取一字节的数据，然后把它写到内存，反之亦然，而且还要插入等待状态。这个控制器只支持 8 位和 16 位传输。

总线主设备 DMA 需要外围卡上有更复杂的电路，但是能快得多地传输数据，而且传输每个字节或字只需要一个周期。总线主设备控制器还可以执行 32 位传输。

内核函数 `set_dma_mode`、`set_dma_addr`、`set_dma_count`、`enable_dma`、`disable_dma`、`request_dma`、`free_dma` 和 `clear_dma_ff` 用于建立 DMA 传输；使用 DMA 的驱动程序必须运行在内核模式。

本章的示例内核驱动程序不使用 DMA。查看本章结尾的参考信息，找到在 Linux 下使用 DMA 的更多信息。

31.5.3 引发使用设备驱动程序的中断

当一个硬件设备需要驱动程序的注意时，它通过引发一个中断来实现。处理器通过存储它自身的状态并跳转到先前注册的中断处理程序来响应一个中断。当中断处理程序返回时，处理器恢复自己的状态并在它离开的地方继续执行。在处理更高优先级的中断时或在临界代码段，中断可以被屏蔽（使无效）。

例如，一个串口 UART 在接收缓冲区快满或发送缓冲区快空时，引发一个中断。处理器通过传输数据响应中断。

中断对用户模式程序不可用。如果你的设备驱动程序需要一个中断处理程序，就必须作为内核模式设备驱动程序来实现。一个中断处理程序函数用下面的语法来声明：

```
#include <linux/inerrupt.h>
#include <linux/sched.h>
void handler(int irq, void *dev_id, struct pt_regs *regs);
```

在大多数情况下你将会忽略这些变量的实际值，但是你的处理程序必须以这种风格来声明（你可以改变函数的名字），否则你会遇到编译错误。第一个参数用于允许一个处理程序处理多个中断并且告诉哪一个被响应了。第二个参数是在你注册这个中断时传递的任意值的一个拷贝（见下面代码）。这个值一般是 NULL，或是你定义的一个结构指针，这个结构用来传递信息给一个可以修改其操作的中断处理程序。再次说明，这有助于用一个处理程序处理多个中断，并且，如果你想要一个更清洁的程序结构的话，它可以帮助你避免使用全局变量和驱动程序通信。这些特性将更有可能被用在一个驱动程序控制多个设备实例

的情况下。`pt_regs` 是被存下来的处理器寄存器，以防你需要检查或修改中断时机器的状态。在一个一般的设备驱动程序中将不需要使用这些。

我们使用下面的调用向内核注册一个中断程序：

```
request_irq(irq, handler, flags, device, dev_id);
```

如果操作成功它返回一个整数 0。第一个参数是被请求中断的号码。这是硬件 IRQ 的号码，不是它们被映射到的软件中断号；注意，由于 PC 机硬件设计的方法，IRQ 2 和 IRQ 9 是古怪的。第二个参数是你的处理程序函数的指针。第三个参数包含着标志。`SA_SHIRQ` 标志告诉内核你想和其他设备驱动程序共享这个中断，假定你的驱动程序、其他驱动程序和硬件支持这个选项。`SA_INTERRUPT` 影响调度程序在中断处理函数返回后是否运行。第四个参数以一个文本串给出驱动程序的名字，这个名字将在 `/proc/interrupts` 中出现。最后一个参数完全不被内核利用，但将在每次处理函数被调用时传递给它。`free_irq` 函数采用一个参数中断号来解除注册你的处理程序。

函数 `cli` (Clear Interrupt enable, 清除中断使能) 禁止中断，而 `sti` (Set Interrupt enable, 设置中断使能) 把它们重新使能。这些函数用来保护对临界数据结构的访问。这些函数通常用来保护上半部和下半部不受对方的影响，或者保护一个中断处理程序不受其他中断处理程序的影响，但是它也可以用来保护其他重要的操作。不应该禁止中断很长时间。

对 `sti` 和 `cli` 的调用可以被嵌套；你调用 `cli` 的次数必须和调用 `sti` 的次数一样多。在示例的驱动程序中，我不需要直接使用这些函数，但我使用的一些内核函数调用了它们以保护被这些函数操纵的数据结构。

31.5.4 设备驱动程序分层

设备驱动程序通常被分成分别叫作下半部和上半部的两层。上半部完成与内核的通信工作，而下半部在需要实际访问硬件时调用。通过把设备驱动程序分成不同的层次，就很容易把实时性要求不高的部分同要求非常实时的部分分隔开。

当系统调用发出时，当前的任务继续执行，但把它的状态改为特权的内核模式操作。如果内核代码决定该调用应当由我们的设备驱动程序处理，它就调用我们设备驱动程序中事先注册的相应函数。这些函数合起来就是这个设备驱动程序的上半部。这些函数通常读取下半部排入队列的数据，把供下半部读取的数据排入队列，改变一个文件的位置，打开一个文件，关闭一个文件，或执行其他类似的操作。上半部通常不直接和设备通信。如果需要的话，上半部函数通常使自己（和当前的任务）进入睡眠，直到下半部把实际的工作做完。

一个驱动程序的下半部处理与硬件设备的实际的通信。下半部通常周期性地被调用以响应来自设备的硬件中断或 100Hz 的系统瞬时时钟（它本身是被硬件中断触发的）。下半部的函数不可以睡眠而应快速地完成它们的工作；如果它们不能执行一个操作而无需等待，它们通常返回并在响应下一次中断时再试一次。

31.6 简单的用户模式测试驱动程序

在编写实际的内核级设备驱动程序之前，最好通过首先在用户模式下部分实现它来试验驱动程序的原理。程序清单 31.1 是一个非常简单的程序，它只是让步进电机缓慢地做一定次数的旋转。这个程序检验单个步进电机（电机 0）的连接正确性，并示范 iopl 和 outb 的用法。

程序清单 31.1 简单的用户模式驱动程序

```
/* Must be compiled with -O for outb to be inlined, */
/* otherwise link error */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <asm/io.h>

int base=0x378;

void verbose_outb(int port, int value)
{
    printf("outb(%02X,%04X)\n",port,value);
    outb(port,value);
}

unsigned char steptable[8]={0x01,0x05,0x04,0x06,0x02,0x0A,0x08,
                           0x09};

slow_sweep()
{
    int i;
    for(i=0;i<800;i++) {
        verbose_outb(steptable[i%8],base+0);
        usleep(100000);
    }
}

main()
{
    int i;
    int inval;
    int outval;

    printf("this program must be run as root\n");
    iopl(3); /* Enable i/o (if root) */
    slow_sweep();
}
```


31.7 创建内核驱动程序

本节概述如何创建一个更完整的作为内核驱动程序运行的步进电机驱动程序。更确切地说，是一个可加载的内核模块。尽管性能将被限制，它也可以被编译为一个用户模式的驱动程序。

31.7.1 查看源代码

本小节将研究示例驱动程序的源代码。许多用来写驱动程序的内核函数将在它们原来的环境中被显示。源代码不得被损坏一点以适应发布者 65 列的限制。特别注意，有些字符串被分为在相邻两行的两小串；编译器将自动把它们连接回单一的一串。

头文件

程序清单 31.2 显示了头文件 `stepper.h`，它定义了通过 `ioctl` 函数向驱动程序发送的一些命令。后面将更深入地讨论 `ioctl`。

程序清单 31.2 `stepper.h`

```
/* define ioctls */

#define STEPPER_SET_DEBUG           0x1641
#define STEPPER_SET_SKIPTICKS      0x1642
#define STEPPER_SET_VERBOSE_IO     0x1643
#define STEPPER_SET_VERBOSE_MOVE   0x1644
#define STEPPER_START               0x1645
#define STEPPER_STOP                0x1646
#define STEPPER_CLEAR_BUFFERS      0x1647
#define STEPPER_NOAUTO              0x1648
#define STEPPER_AUTO                0x1649
```

环形缓冲区头文件

程序清单 31.3 显示了环形缓冲区代码的头文件。程序清单 31.4 显示了环形缓冲区的实现。这个模块实现了一个环形缓冲区或 FIFO（先入先出，First In First Out）缓冲区。这些环形缓冲区用于在上半部和下半部之间缓冲数据。它们在单前的驱动程序中被删除了，但对编写需要更多缓冲区的更正式的驱动程序有用。这些函数必须是可重新输入的，这样函数可同时运行许多次，不会产生负面影响。下半部可能中断上半部的执行。

这两个文件定义了数据结构、一个初始化函数和两个函数来写（排队）和读（出队）数据。现在它们只接受一次一字节的读写，尽管调用约定不需要为多字节传输而改变。

程序清单 31.3 `ring.h`

```
#define RING_SIGNATURE 0x175DE210

typedef struct {
    long signature;
    unsigned char *head_p;
```

```

    unsigned char *tail_p;
    unsigned char *end_p;    /* end + 1 */
    unsigned char *begin_p;
    unsigned char buffer[32768];
} ring_buffer_t;

extern void ring_buffer_init(ring_buffer_t *ring);
extern int ring_buffer_read(ring_buffer_t *ring,
    unsigned char *buf, int count);
extern int ring_buffer_write(ring_buffer_t *ring,
    unsigned char *buf, int count);

```

程序清单 31.4 ring.c

```

/* Copyright 1998,1999, Mark Whitis <whitis@dbd.com> */
/* http://www.freelabs.com/~whitis/ */

#include "ring.h"

int ring_debug=1;

#ifdef TEST
#define printk printf
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#endif

void ring_buffer_init(ring_buffer_t *ring)
{
    ring->signature=RING_SIGNATURE;
    ring->head_p=ring->buffer;
    ring->tail_p=ring->buffer;
    ring->begin_p=ring->buffer;
    ring->end_p=&ring->buffer[sizeof(ring->buffer)];
    #if 0
        strcpy(ring->buffer,"This is a test.\n");
        ring->head_p +=16;
    #endif
}

/*
 * returns number of bytes read. Will not block (blocking will
 * be handled by the calling routine if necessary).
 * If you request to read more bytes than are currently
 * available, it will return
 * a count less than the value you passed in
 */
int ring_buffer_read(ring_buffer_t *ring, unsigned char *buf,
    int count)
{

```

```

#ifdef PARANOID
    if(ring_debug>5) {
        printk("das1600: ring_buffer_read(%08X,%08X,%d)\n",
            ring,buf,count);
    }
    if(ring->signature != RING_SIGNATURE) {
        printk("ring_buffer_read: signature corrupt\n");
        return(0);
    }
    if(ring->tail_p < ring->begin_p) {
        printk("ring_buffer_read: tail corrupt\n");
        return(0);
    }
    if(ring->tail_p > ring->end_p) {
        printk("ring_buffer_read: tail corrupt\n");
        return(0);
    }
    if(count != 1) {
        printk("ring_buffer_read: count must currently be 1\n");
        return(0);
    }
#endif;
if(ring->tail_p == ring->end_p) {
    ring->tail_p = ring->begin_p;
}
if(ring->tail_p == ring->head_p) {
    if(ring_debug>5) {
        printk("ring_buffer_read: buffer underflow\n");
    }
    return(0);
}
*buf = *ring->tail_p++;
return(1);
}
/*
 * returns number of bytes written. Will not block (blocking
 * will be handled by the calling routine if necessary).
 * If you request to read more bytes than are currently
 * available, it will return
 * a count less than the value you passed in
 */
int ring_buffer_write(ring_buffer_t *ring, unsigned char *buf,
    int count)
{
    unsigned char *tail_p;
#ifdef PARANOID

```

```
if(ring->signature != RING_SIGNATURE) {
    printk("ring_buffer_write: signature corrupt\n");
    return(0);
}
if(ring->head_p < ring->begin_p) {
    printk("ring_buffer_write: head corrupt\n");
    return(0);
}
if(ring->head_p > ring->end_p) {
    printk("ring_buffer_write: head corrupt\n");
    return(0);
}
if(count != 1) {
    printk("ring_buffer_write: count must currently be 1\n");
    return(0);
}
#endif
/* Copy tail_p to a local variable in case it changes */
/* between comparisons */
tail_p = ring->tail_p;
if( (ring->head_p == (tail_p - 1) )
|| ((ring->head_p == (ring->end_p - 1)) && (tail_p==ring->begin_p)
    ) ) {
    if(ring_debug>5) {
        printk("ring_buffer_write: buffer overflow\n");
    }
    return(0);
}

*ring->head_p++ = *buf;

if(ring->head_p == ring->end_p ) {
    ring->head_p = ring->begin_p;
}
return(1);
}

#ifdef TEST
ring_buffer_t buffer;
main()
{
    char c;
    char c2;
    int child;
    int rc;
    int i;
    int j;
    char lastread;
```

```

    int errors;
    int reads;
    int writes;

    ring_buffer_init(&buffer);

    c=0;
    lastread=-1;
    errors=0;
    reads=0;
    writes=0;
    for(j=0; j<50000; j++) {
        for(i=0; i<31; i++) {
            rc=ring_buffer_write(&buffer, &c, 1);
            writes++;
            if(ring_debug>2) {
                printf("ring_buffer_write returned %d,"
                    "was passed %d\n",rc,c);
            }
            if(rc==1) c++;
        }

        for(i=0; i<47; i++) {
            rc=ring_buffer_read(&buffer, &c2, 1);
            reads++;
            if(ring_debug>2) {
                printf("ring_buffer_read returned: rc=%d,c2=%d\n",rc,c2);
            }
            if(rc==1) {
                if(c2!=(char)(lastread+1)) {
                    printf("ERROR: expected %d, got %d\n",
                        (char)(lastread+1),c2);
                    errors++;
                }
                lastread=c2;
            }
        }
    }
    printf("number of errors=%d\n",errors);
    printf("number of reads=%d\n",reads);
    printf("number of writes=%d\n",writes);
}
#endif

```

预备代码

程序清单 31.5 显示了预备代码。其中包括所需的#include 指令和变量声名。一些变量在本章后面的“使用内核驱动程序”小节中描述。这个程序如果作为内核级驱动程序被编译，预处理程序宏__KERNEL__将被定义；注意这将影响许多系统头文件以及驱动程序源

码中的控制条件编译。

程序清单 31.5 预备代码

```

/* Copyright 1999 by Mark Whitis. All rights Reserved */

/* Must be compiled with -O for outb to be inlined, otherwise */
/* link error */
/* Usermode: gcc -g -O -o teststep teststep.c */
/* LKM: */
/* gcc -O -DMODULE -D__KERNEL__ -o stepperout.o -c stepper.c */
/* ld -r -o stepper.o stepperout.o ring.o */

#include "stepper.h"

#ifdef __KERNEL__
#include <linux/kernel.h>
#include <linux/config.h>
#ifdef MODULE
#include <linux/module.h>
#include <linux/version.h>
#else
/* This code is GNU copylefted and should not be statically */
/* linked with the kernel. */
#error This device driver must be compiled as a LKM
#define MOD_INC_USE_COUNT
#define MOD_DEC_USE_COUNT
#endif
#include <linux/types.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/errno.h>
#include <asm/segment.h>
#include <asm/io.h>
#include <linux/sched.h>
#include <linux/tqueue.h>
#else
#include <unistd.h>
#include <stdlib.h>
/* including stdio will cause grief */
#include <stdio.h>
#include <asm/io.h>
#endif

#if 0
#include <stdarg.h>
#include <ctype.h>
#endif
#include <string.h>

```

```

#ifdef __KERNEL__
#include "ring.h"
ring_buffer_t read_buffer;
ring_buffer_t write_buffer;
/* Parallel port interrupt to use for timing */
/* 0=use Linux 100hz jiffie timer */
static int irq = 0;
/* Major device # to request */
static int major = 31;
/* The device # we actually got */
static int stepper_major = 0;

struct wait_queue *read_wait = NULL ;
/* used to block user read if buffer underflow occurs */
struct wait_queue *write_wait = NULL;
/* used to block user write if buffer overflow occurs */
#endif

static int debug = 1;
static int verbose_io=0;
static int verbose_move=0;
static int base=0x378;
static int power_down_on_exit=1;

/* The following set the delay between steps */
#ifdef __KERNEL__
static int delay = 50000; /* for usleep */
static int fastdelay = 0; /* delay loop */
#else
static int skipticks = 0;
#endif

/* the following value can be set to 0 to disable the */
/* actual I/O operations. This will allow experimenting */
/* on a system which does not have a free parallel port */
static int do_io = 1;

#ifdef __KERNEL__
static int read_is_sleeping = 0;
static int write_is_sleeping = 0;
static int abort_read = 0;
static int abort_write = 0;
static int read_is_open = 0;
static int write_is_open = 0;
static int interrupts_are_enabled = 0;
static int autostart=1;
#endif

static int tick_counter=0;

int xdest = 0;

```

```

int xuser = 0;
int xpos = 0;

int ydest = 0;
int yuser = 0;
int ypos = 0;

int zdest = 0;
int zuser = 0;
int zpos = 0;

unsigned short word;

```

verbose_outb

程序清单 31.6 显示了一个 outb 函数的包装，它提供了可选的调试输出，并为缺少必要硬件的实验提供了禁止 outb 的能力。

程序清单 31.6 verbose_outb

```

void verbose_outb(int port, int value)
{
    #ifndef __KERNEL__
        if(verbose_io) printf("outb(%02X,%04X)\n",port,value);
    #else
        if(verbose_io) printk("outb(%02X,%04X)\n",port,value);
    #endif

    if(do_io) outb(port,value);
}

```

延迟函数

程序清单 31.7 所示的 do_delay 函数只在用户模式驱动程序中使用，它提供步进之间的延迟。它将使用 usleep 库函数除非 fastdelay 非 0，这种情况下它将使用一个延迟循环。这些延迟只设置最少的延迟；在用户模式的程序中，多任务将引起额外的延迟。

程序清单 31.7 延迟函数

```

#ifdef __KERNEL__
    void do_delay()
    {
        ;
    }
#else
    void do_delay()
    {
        int i;

        if(fastdelay>0) {
            for(i=0;i<fastdelay;i++) {
                /* dummy operation so optimizer doesn't */
            }
        }
    }

```



```

        /* eliminate loop */
        verbose_outb( (word&0x00FF),base+0);
    }
} else {
    usleep(delay);
}
}
#endif

```

状态报告

程序清单 31.8 显示了一个函数，它报告步进电机在每个轴的位置。在用户模式程序中，这将被显示到 `stdout`；在内核驱动程序中，这些结果将被存储在环形缓冲区中供用户模式控制程序用 `read`、`fgets`、`fscanf` 或类似的函数读取。16 位输出字的值也被报告；这是最近输出到并口数据和状态寄存器的值。

程序清单 31.8 状态报告

```

void report_status()
{
    char buf[128];
    int rc;
    int i;
    int len;

    sprintf(buf, "x=%d,y=%d,z=%d,word=%08X\n", xpos, ypos, zpos,
        word);
#ifdef __KERNEL__
    len=strlen(buf);

    for(i=0;i<len;i++) {
        rc=ring_buffer_write(&read_buffer,&buf[i],1);
    }
    /* we need to wake up read function */
    if(read_is_sleeping) {
        if(debug>=5) printk("stepper: waking up read/n");
        wake_up_interruptible(&read_wait);
    }
#else
    puts(buf);
#endif
}

```

步进控制

程序清单 31.9 给出了实际驱动步进电机的函数。函数 `move_one_step` 将把每个坐标轴向前移动一个半步。因为驱动程序的下半部每当需要移动一步时就被调用一次，这需要是一个简单的增加的移动，而不是完整的端到端的移动。在用户模式版本中，`move_all` 将用

来做一个完整的移动。

数组 `x_steptable`, `y_steptable` 和 `z_steptable` 指出, 为每次电动旋转的八个半步中的每一个半步, 需要设置哪些位。这些位可以被改变以适应不同的连线或反转电机旋转的“前向”。变量 `flipbits` 用于反转某些位以补偿包含在标准并口接口中的反向器。变量 `irq_enable_bit` 以后将被用于使能并口卡上的中断。

函数 `parse_one_char` 实现一个移动指令的简单解析。它以每次一个字符的方式运行, 下半部将不需要把命令重新组织成多行来使用这个函数。

程序清单 31.9 步进控制

```
unsigned short x_steptable[8]=
    {0x0001,0x0005,0x0004,0x0006,0x0002,0x000A,0x0008,0x0009};
unsigned short y_steptable[8]=
    {0x0010,0x0050,0x0040,0x0060,0x0020,0x00A0,0x0080,0x0090};
unsigned short z_steptable[8]=
    {0x0100,0x0500,0x0400,0x0600,0x0200,0x0A00,0x0800,0x0900};
unsigned short flipbits = 0x0B00;
unsigned short irq_enable_bit=0x1000;

void move_one_step()
{
    /* This is a very simple multiaxis move routine */
    /* The axis will not move in a coordinate fashion */
    /* unless the angle is a multiple of 45 degrees */
    if(xdest > xpos) {
        xpos++;
    } else if(xdest < xpos) {
        xpos--;
    }

    if(ydest > ypos) {
        ypos++;
    } else if(ydest < ypos) {
        ypos--;
    }

    if(zdest > zpos) {
        zpos++;
    } else if(zdest < zpos) {
        zpos--;
    }

    word = x_steptable[xpos%8]
        | y_steptable[ypos%8]
        | z_steptable[zpos%8];

#ifdef __KERNEL__
    if(interrupts_are_enabled) word |= irq_enable_bit;
#endif
}
```

```

/* Some of the signals are inverted */
word ^= flipbits;

/* output low byte to data register */
verbose_outb( (word & 0x00FF), base+0);
/* output high byte to control register */
verbose_outb( (word >> 8), base+2);

if(verbose_move) report_status();
}

void move_all()
{
    while( (xpos!=xdest) || (ypos!=ydest) || (zpos!=zdest) ) {
        move_one_step();
        do_delay();
    }
}

void parse_one_char(char c)
{
    static int value;
    static int dest=' ';
    static int negative=0;

    #if 0
        c = toupper(c);
    #else

        if( (c>'a') && (c<'z') ) { c = c - 'a' + 'A'; }
    #endif

    switch(c) {
        case('X'):
        case('Y'):
        case('Z'):
            dest=c;
            break;
        case('='):
            break;
        case('-'):
            negative = !negative;
            break;
        case('+'):
            negative = 0;
        case('0'):
        case('1'):
        case('2'):
        case('3'):
        case('4'):

```

```

        case('5'):
        case('6'):
        case('7'):
        case('8'):
        case('9'):
            value *= 10;
            value += (c-'0');
            break;
        case('?'):
            report_status();
        case('\r'):
        case('\n'):
        case(','):
            if(negative) {
                value = -value;
            }
            if(dest=='X') {
                xuser = value;
            } else if(dest=='Y') {
                yuser = value;
            } else if(dest=='Z') {
                zuser = value;
            }
            value = 0;
            negative = 0;

            if( (c=='\n') || (c=='\r') ) {
                xdest = xuser;
                ydest = yuser;
                zdest = zuser;

                #ifdef __KERNEL__
                if(debug>=3) {
                    printk("xdest=%d ydest=%d zdest=%d\n",
                        xdest,ydest,zdest);
                }
                #endif
            }
            break;
    }
}

```

定位操作

函数 `stepper_lseek`，如程序清单 31.10 所示，实际上是第一个针对内核级驱动程序并实现上半部接口的函数。这个函数将在发出 `lseek` 调用设定文件位置时被调用。注意 `open`、`seek`、`fopen` 和 `fseek` 也可能调用 `lseek`。这种情况下我们不能改变文件位置，所以我们返回 0 值。

参数 `inode` 和 `file` 提供了指向内核内部数据结构的指针, 这些结构定义了用户已经打开的文件并将被传递给所有的上半部的函数。它们在 `/usr/include/linux/fs.h` 中定义。

参数 `offset` 和 `orig` 定义了偏移量和起始地址 (文件的开始、文件的结束或当前位置)。这些值用于设定当前文件的位置, 在 `lseek` 的手册页中提供了关于这些值的更详细的文档。

程序清单 31.10 定位操作

```
#ifdef __KERNEL__
static int stepper_lseek(struct inode *inode,
                        struct file *file, off_t offset, int orig)
{
    /* Don't do anything, other than set offset to 0 */
    return(file->f_pos=0);
}
#endif
```

读写操作

程序清单 31.11 给出了 `stepper_read` 函数, 这是实现文件读操作的上半部函数。在我们的例子中, 这个函数将被用来读取由 `report_status` 放置在读环形缓冲区中的状态响应。这个函数将在用户空间对设备驱动程序控制的文件发出 `read` 系统调用时被调用; `fread`、`fgets`、`fscanf`、`fgetc` 和其他库函数调用这个系统调用。

参数 `node` 和 `file` 与前面描述的 `stepper_lseek` 的前两个参数相同。其他两个参数, `buf` 和 `count`, 提供了供填充的一个数据缓冲区的地址和要读取的字节数。内核函数 `verify_area` 必须被调用以验证缓冲区是可用的。

这个函数调用 `ring_buffer_read` 得到数据。如果数据不可用, 它必须睡眠。它用内核函数 `interruptible_sleep_on` 使自己 (和调用的任务) 进入休眠状态, 并把自己加入 `read_wait` 任务队列, 其中的任务 (这个情况下只有一个) 在又有数据可用时, 被下半部函数唤醒。`read_wait` 队列已在本章前面的“预备代码”中定义; 我们可以定义任意数量的队列。如果你想把多个任务放入等待队列, 则需要分配更多的 `wait_queue` 项, 并把它们用 `next` 成员链接在一起。等待队列在 `/usr/include/linux/sched.h` 中声明。函数 `interruptible_sleep_on` 在 `/usr/include/linux/sched.h` 中声明, 而实际函数的源代码在 `/usr/src/linux/kernel/sched.c` 中。

如上所述, `stepper_read` 和 `stepper_write` 函数也唤醒对方; 这可能不必要, 因为下半部应当做这些, 但这有助于防止读和写之间的死锁。设置变量 `read_is_sleeping` 以告诉下半部在又有数据可用的时候唤醒我们。

函数 `stepper_write` 执行和 `stepper_read` 相反的操作, 并且除了数据传输的方向被反转了之外看起来相当类似。如果发出的 `write` 系统调用带有文件描述符 (此文件描述符引用由我们的设备控制的文件), 则调用这个函数。

程序清单 31.11 读写操作

```
#ifdef __KERNEL__
static int stepper_read(struct inode *node,
                      struct file *file, char *buf, int count)
```

```
{
    static char message[] = "hello, world\n";
    static char *p = message;
    int i;
    int rc;
    char xferbuf;
    int bytes_transferred;
    int newline_read;

    newline_read=0;

    if(!read_is_open) {
        printk("stepper: ERROR: stepper_read() called while "
               "not open for reading\n");
    }
    bytes_transferred=0;

    if(debug>2) printk("stepper_read(%08X,%08X,%08X,%d)\n",
                      node,file,buf,count);
    if(rc=verify_area(VERIFY_WRITE, buf, count) < 0 ) {
        printk("stepper_read(): verify area failed\n");
        return(rc);
    }
    for(i=count; i>0; i--) {
        #if 0
            if(!*p) p=message;
            put_fs_byte(*p++,buf++);
        #else
            while(1) {
                rc=ring_buffer_read(&read_buffer,&xferbuf,1);

                if(debug>3) {
                    printk(
                        "stepper: ring_buffer_read returned %d\n",
                        rc);
                }
                if(rc==1) {
                    bytes_transferred++;
                    put_fs_byte(xferbuf,buf++);
                    if(xferbuf=='\n') {
                        printk("stepper_read(): newline\n");
                        newline_read=1;
                    }
                    break; /* read successful */
                }
            }

            read_is_sleeping=1;
            if(debug>=3) printk("stepper: read sleeping\n");
            interruptible_sleep_on(&read_wait);
        }
    }
}
```

```

        read_is_sleeping=0;
        if(abort_read) return(bytes_transferred);
    }
    /* we want read to return at the end */
    /* of each line */
    if(newline_read) break;
#endif
}
if(write_is_sleeping) wake_up_interruptible(&write_wait);
if(debug>=3) {
    printk("stepper_read(): bytes=%d\n", bytes_transferred);
}
return(bytes_transferred);
}

static int stepper_write(struct inode *inode,
    struct file *file, const char *buf, int count)
{
    int i;
    int rc;
    char xferbuf;
    int bytes_transferred;

    if(!write_is_open) {
        printk("stepper: ERROR: stepper_write() called"
            " while not open for writing\n");
    }

    bytes_transferred=0;

    if(rc=verify_area(VERIFY_READ, buf, count) < 0 ) {
        return(rc);
    }

    for(i=count; i>0; i--) {
        get_fs_byte(xferbuf,buf++);
        while(1) {
            rc=ring_buffer_write(&write_buffer,&xferbuf,1);
            if(rc==1) {
                bytes_transferred++;
                break;
            }
            if(debug>10) printk("stepper: write sleeping\n");
            write_is_sleeping=1;
            interruptible_sleep_on(&write_wait);
            write_is_sleeping=0;
            if(abort_write) return(bytes_transferred);

```

```

    }
}
if(read_is_sleeping) wake_up_interruptible(&read_wait);
return(bytes_transferred);
}
#endif

```

ioctl 命令

如程序清单 31.12 所示，每当引用驱动程序所控制文件的文件描述符的系统调用 `ioctl` 发出时，就调用函数 `stepper_ioctl`。根据 `ioctl` 的手册页，`ioctl` 是“那些不是很好符合 UNIX 流 I/O 模型的操作的总容器”。例如，`ioctl` 通常用于设定串口上的波特率和其他通信参数以及执行许多 TCP/IP 套接口的操作。以太网和其他网络的接口的用 `ifconfig` 设定的各种参数，像 ARP 表项一样，也使用 `ioctl` 命令控制。在我们这种情况下，将使用它们来改变调试变量的值以及改变移动的速度。将来，它们也将被用来控制开始和停止中断（及设备的动作）。我有几分随机地给各个 `ioctl` 指定了号码，以避免在 `/usr/include/ioctls.h` 中定义的值。驱动程序还接受 `TCGETS`（见第 22 章），我们简单地把它忽略了；清空 `arg` 指向的数据结构或返回一个 `ENOTTY` 错误可能更合适。

程序清单 31.12 `ioctl` 命令

```

void stepper_start()
{
    /* do nothing at the moment */
}

void stepper_stop()
{
    /* do nothing at the moment */
}

static int stepper_ioctl(struct inode *iNode,
    struct file *filePtr, unsigned int cmd, unsigned long arg)
{
    switch(cmd) {
        case(STEPPER_SET_DEBUG):
            /* unfriendly user might crash system */
            /* by setting debug too high. Only allow */
            /* root to change debug */
            if((current->uid==0) || (current->euid==0)) {
                debug=arg;
            } else {
                return(EPERM);
            }
            break;

        case(STEPPER_SET_SKIPTICKS):
            skipticks=arg;

```



```

        break;

case (STEPPER_SET_VERBOSE_IO) :
    verbose_io=arg;
    break;

case (STEPPER_SET_VERBOSE_MOVE) :
    verbose_move=arg;
    break;

case (TCGETS) :
    break;

#if 0
    /* autostart and start/stop are not implemented */
    /* these would be used to enable interrupts */
    /* only when the driver is in use and to */
    /* allow a program to turn them on and off */

case (STEPPER_START) :
    if(debug) printk("stepper_ioctl(): start\n");
    stepper_start();
    break;

case (STEPPER_STOP) :
    if(debug) printk("stepper_ioctl(): stop\n");
    stepper_stop();
    break;

case (STEPPER_CLEAR_BUFFERS) :
    if(debug) {
        printk("stepper_ioctl(): clear buffers\n");
    }
    ring_buffer_init(&read_buffer);
    ring_buffer_init(&write_buffer);
    break;

case (STEPPER_AUTO) :
    if(debug) {
        printk("stepper_ioctl(): enable autostart\n");
    }
    autostart = 1;
    break;

case (STEPPER_NOAUTO) :
    if(debug) {
        printk("stepper_ioctl(): disable autostart\n");
    }
    autostart = 0;
    break;
#endif

default:
    printk("stepper_ioctl(): Unknown ioctl %d\n",cmd);

```

```

        break;
    }
    return(0);
}
#endif

```

打开与关闭操作

程序清单 31.13 显示了 `stepper_open` 和 `stepper_release` 函数, 它们执行打开和关闭操作。这些函数在对设备驱动程序所控制的文件的 `open` 或 `close` 系统调用发出时被调用。在这个例子中, 我们控制两个字符特殊文件: 分别有副设备号 0 和 1 的 `/dev/stepper` 和 `/dev/stepper_ioctl`。因为不允许任意数目的进程打开和关闭这些设备, 我们不需要了解哪些流和哪个进程相联系。允许同时打开三个: 一个只为写的打开, 一个只为读的打开和一个(或多个) 只为 `ioctl` 的打开。

程序清单 31.13 打开和关闭操作

```

#ifdef __KERNEL__
static int stepper_open(struct inode *inode,
    struct file *file)
{
    int rc;
    int minor;

    minor = MINOR(inode->i_rdev);
    printk("stepper: stepper_open() - file->f_mode=%d\n",
        file->f_mode);

    /*
     * As written, only one process can have the device
     * open at once. For some applications, it might be
     * nice to have multiple processes (one controlling, for
     * example, X and Y while the other controls Z).
     * I use two separate entries in /dev/, with
     * corresponding minor device numbers, to allow a
     * program to open for ioctl while another program
     * has the data connection open.
     * This would allow a Panic Stop application to be
     * written,
     * for example.
     *   Minor device = 0 ' read and write '
     *   Minor device = 1 ' ioctl() only '
     */

    /* if minor!=0, we are just opening for ioctl */
    if(minor!=0) {
        MOD_INC_USE_COUNT;
        return(0);
    }
}

```

```

    }

    if(autostart) stepper_start();

    if(file->f_mode==1) {
        /* read */
        if(read_is_open) {
            printk("stepper: stepper_open() - read busy\n");
            return(-EBUSY);
        } else {
            read_is_open=1;
            abort_read=0;
            MOD_INC_USE_COUNT;
        }
    } else if(file->f_mode==2) {
        /* write */
        if(write_is_open) {
            printk("stepper: stepper_open() - write busy\n");
            return(-EBUSY);
        } else {
            write_is_open=1;
            abort_write=0;
            MOD_INC_USE_COUNT;
        }
    } else {
        printk("stepper: stepper_open() - unknown mode\n");
        return(-EINVAL);
    }

    if(debug) printk("stepper: stepper_open() - success\n");
    return(0);
}

void stepper_release(struct inode *inode, struct file *file)
{
    int minor;

    minor = MINOR(inode->i_rdev);
    if(minor!=0) {
        MOD_DEC_USE_COUNT;
        return;
    }

    printk("stepper: stepper_release() - file->f_mode=%d\n",
        file->f_mode);
    if(file->f_mode==1) {
        /* read */
        if(read_is_open) {
            abort_read=1;
            if(read_is_sleeping) {

```

```

        wake_up_interruptible(&read_wait);
    }
    read_is_open=0;
    MOD_DEC_USE_COUNT;
} else {
    printk("stepper: ERROR: stepper_release() "
           "called unexpectedly (read)\n");
}
} else if(file->f_mode==2) {
    /* write */
    if(write_is_open) {
        abort_write=1;
        if(write_is_sleeping) {
            wake_up_interruptible(&write_wait);
        }
        write_is_open=0;
        MOD_DEC_USE_COUNT;
    } else {
        printk("stepper: ERROR: stepper_release() called"
               " unexpectedly (write)\n");
    }
} else {
    printk("stepper: stepper_release() "
           "- invalid file mode\n");
}

if(!read_is_open && !write_is_open) {
    stepper_stop();
}
}
#endif

```

文件操作结构

这个结构，如程序清单 31.14 所示，有指向我们前面所定义的上半部所有函数的指针。我们还可以定义其他一些函数，但我们将把它们设为 NULL，而内核将使用默认的处理程序。可能实现一个 `stepper_fsync` 函数会有意义，这样我们可以在我们的用户程序中调用 `fflush`（它调用 `fsync`），以避免用户程序太超前于驱动程序。该函数将睡眠直到 `write_buffer` 空及步进电机完成它们的最近一次移动。

程序清单 31.14 文件操作结构

```

static struct file_operations stepper_fops = {
    stepper_lseek,           /* lseek */
    stepper_read,           /* read */
    stepper_write,          /* write */
    NULL,                   /* readdir */
    NULL,                   /* select */

```

```

    stepper_ioctl,          /* ioctl */
    NULL,                   /* mmap */
    stepper_open,           /* open */
    stepper_release,        /* close */
    NULL,                   /* fsync */
    NULL,                   /* fasync */
    NULL,                   /* check_media_change */
    NULL,                   /* revalidate */
};

```

下半部

程序清单 31.15 中的函数和前面在程序清单 31.9 中显示的实际步进电机的控制函数一起，实现了设备驱动程序的下半部。根据是使用中断还是定时器报时（瞬时时钟），`interrupt_handler` 或 `timer_tick_handler` 将被调用以响应定时器报时或硬件中断。无论在那种情况我们都要做同样的事情，所以简单地调用另一个函数做这项工作，我们称它为 `bottom_half`。

如果 `cleanup_module` 在等待我们从定时器报时队列中删除自己（这是通过处理下一次报时而不把自己加回定时器报时队列来实现的），则我们只是简单地唤醒 `cleanup_module`，并不做其他任何事情。如果步进电机还没有处理最近一次移动，我们将读取所有的由 `stepper_write` 送入队列的字符，并把它们一次一个地解析，直到它们引起一次移动。如果 `write` 由于环形缓冲区满了而在睡眠，我们将唤醒它，因为它可能能够写更多的字符。

调用 `move_one_step` 来做和设备进行交互的实际工作。如果使用定时器报时，我们每次必须把自己放回定时器报时队列。

程序清单 31.15 下半部

```

static int using_jiffies = 0;

static struct wait_queue *tick_die_wait_queue = NULL;

/* forward declaration to resolve circular reference */
static void timer_tick_handler(void *junk);

static struct tq_struct tick_queue_entry = {
    NULL, 0, timer_tick_handler, NULL
};

static void bottom_half()
{
    int rc;
    char c;

    tick_counter++;
    if(tick_die_wait_queue) {
        /* cleanup_module() is waiting for us */
        /* Don't reschedule interrupt and wake up cleanup */
        using_jiffies = 0;
    }
}

```

```

        wake_up(&tick_die_wait_queue);
    } else {
        if((skipticks==0) || ((tick_counter % skipticks)==0)) {
            /* Don't process any move commands if */
            /* we haven't finished the last one */
            if( (xdest==xpos)
                && (ydest==ypos)
                && (zdest==zpos) ) {
                /* process command characters */
                while(
                    (rc=ring_buffer_read(&write_buffer,&c,1))!=1
                ){
                    parse_one_char(c);
                    if((xdest!=xpos)
                       || (ydest!=ypos)
                       || (zdest!=zpos) ) {
                        /* parse_one_char() started a move; */
                        /* stop reading commands so current move*/
                        /* can complete first. */
                        break;
                    }
                }
                if(write_is_sleeping) {
                    wake_up_interruptible(&write_wait);
                }
            }
        }

        move_one_step();

        /* put ourselves back in the queue for the next tick*/
        if(using_jiffies) {
            queue_task(&tick_queue_entry, &tq_timer);
        }
    }
}

static void timer_tick_handler(void *junk)
{
    /* let bottom_half() do the work */
    bottom_half();
}

void interrupt_handler(int irq, void *dev_id,
    struct pt_regs *regs)
{
    /* let bottom_half() do the work */
    bottom_half();
}

```

```

    }
#endif

```

模块初始化与终止

程序清单 31.16 显示了 `init_module` 和 `cleanup_module`，它们是模块的初始化和终止函数。这些函数必须使用这些名字，因为 `insmod` 程序根据名字调用它们。

`init_module` 函数先初始化那两个用来缓冲读写的环形缓冲区，并重置表明读或写函数正在睡眠的标志。它然后用 `register_chrdev` 向系统注册主设备号。这个主设备号必须还没被使用，并且必须和我们在构造设备特殊文件 `/dev/stepper` 和 `/dev/stepper_ioctl` 时使用的号码相匹配。第一个参数是主设备号。第二个是一个用来和主设备号一起在 `/proc/devices` 中列出的标志串。第三个参数是我们前面定义的文件操作结构，在这里我们告诉系统如何调用我们上半部的函数。

我们调用 `check_region` 来测试 I/O 端口是否可用（不被其他驱动程序使用），然后用 `request_region` 注册它们。这些调用以一个基地址和连续的 I/O 地址的计数为参数。如果这个区域可用的话，`check_region` 函数返回 0。这个区域后来将用 `release_region` 释放，它使用同样的参数。

接下来，依据所选的操作模式，我们把下半部函数之一排入队列以接收中断或定时器报时。`request_irq` 函数我们前面描述过了。如果我们使用定时器报时，我们就用 `queue_task()` 把自己放入合适的队列以接收定时器的报时。如果我们使用来自并口卡的硬件中断，就让卡声明硬件中断行，以便我们可以接收来自一个外部硬件时钟源的中断。

为了测试目的，我让驱动程序发起 400 计数的移动（我打开正使用的电机）；在一个产品驱动程序中这应该被注释掉。最后，我们宣布成功地加载并返回了加载的 `insmod` 或 `kmod` 程序。

`cleanup_module` 函数被调用来结束和卸载这个内核模块。它由 `rmmod` 或 `kmod` 程序引发。这个函数基本上把 `init_module` 的动作反过来。内核函数 `unregister_chrdev` 将解除注册先前注册的主设备号，而 `free_irq` 将释放可能已经注册的中断。

显然，没有办法把我们自己从定时器报时队列中删除，所以我们睡眠直到下半部通过接收一个定时器报时而没有再次注册来有效地把我们删除。注意，我应该在发生 `MOD_IN_USE` 事件或者 `unregister_chrdev` 失败时使用 `sleep_on` 技巧。因为我们不能终止该模块的卸载，我们可以永远等待。也可以让下半部不时地唤醒我们来重试失败的操作。如果合适的话，我们将在宣称终止及返回之前把步进电机断电（通过关闭所有的晶体管）。

程序清单 31.16 初始化和终止

```

int init_module(void)
{
    int rc;
    ring_buffer_init(&read_buffer);
    ring_buffer_init(&write_buffer);
    read_is_sleeping=0;
    write_is_sleeping=0;

```

```

    if ((stepper_major = register_chrdev(major, "step",
    &stepper_fops)) < 0 ) {
        printk("stepper: unable to get major device\n");
        return(-EIO);
    }
    /* register_chrdev() does not return the major device */
    /* number, workaround will not be correct if major=0 */
    stepper_major = major;
    if(debug) printk("stepper:init_module():stepper_major=%d\n",
        stepper_major);

    if(check_region(base,4)) {
        printk("stepper: port in use");
        unregister_chrdev(stepper_major, "step");
        return;
    }
    request_region(base,4);

    if(irq && (!interrupts_are_enabled)) {
        rc=request_irq(irq,interrupt_handler,0,"stepper",NULL);
        if(rc) {
            printk("stepper: stepper_start() - request_irq() "
                "returned %d\n",rc);
        } else {
            printk("stepper: stepper_start() "
                "- enabled interrupts\n");
            interrupts_are_enabled = 1;

            /* now that we are ready to receive them */
            /* enable interrupts on parallel card */
            word = irq_enable_bit;
            verbose_outb( (word >> 8), base+2);
        }
    }

    if(!irq) {
        using_jiffies = 1;
        queue_task(&tick_queue_entry, &tq_timer);
    }

    /* give ourselves some work to do */
    xdest = ydest = zdest = 400;

    if(debug) printk( "stepper: module loaded\n");
    return(0);
}

void cleanup_module(void)
{
    abort_write=1;
    if(write_is_sleeping) wake_up_interruptible(&write_wait);
}

```



```

abort_read=1;
if(read_is_sleeping) wake_up_interruptible(&read_wait);

#if 1
    /* Delay 1s for read and write to exit */
    current->state = TASK_INTERRUPTIBLE;
    schedule_timeout(jiffies+100*HZ);
#endif

release_region(base,4);

if(MOD_IN_USE) {
    printk("stepper: device busy, remove delayed\n");
    return;
}

printk("unregister_chrdev(%d,%s)\n",stepper_major, "step");
if( unregister_chrdev(stepper_major, "step") ) {
    printk("stepper: unregister_chrdev() failed.\n");
    printk("stepper: /proc/devices will cause core dumps\n");
    /* Note: if we get here, we have a problem */
    /* There is still a pointer to the name of the device */
    /* which is in the address space of the LKM */
    /* which is about to go away and we cannot abort
    /* the unloading of the module */
}

/* note: we need to release the interrupt here if */
/* necessary otherwise, interrupts may cause kernel */
/* page faults */
if(interrupts_are_enabled) {
    /* Disable interrupts on card before we unregister */
    word &= ~irq_enable_bit;
    verbose_outb( (word >> 8), base+2);

    free_irq(irq,NULL);
}

if(using_jiffies) {
    /* If we unload while we are still in the jiffie */
    /* queue, bad things will happen. We have to wait */
    /* for the next jiffie interrupt */
    sleep_on(&tick_die_wait_queue);
}

if(power_down_on_exit) {
    word=flipbits;
    /* output low byte to data register */
    verbose_outb( (word & 0x00FF), base+0);
    /* output high byte to control register */
    verbose_outb( (word >> 8), base+2);
}

```

```

        if(debug) printk("stepper: module unloaded\n");
    }
#endif

```

main 函数

main 函数，如程序清单 31.17 所示，在内核模式驱动程序中将完全不被使用。它在以用户模式驱动程序编译时用作程序的主入口点。

程序清单 31.17 main 函数

```

#ifndef __KERNEL__
main()
{
    char c;

    /* unbuffer output so we can see in real time */
    setbuf(stdout, NULL);

    printf("this program must be run as root\n");
    iopl(3); /* Enable i/o (if root) */

    printf("Here are some example motion commands:\n");
    printf("  X=100,Y=100,Z=50\n");
    printf("  X=100,Y=100\n");
    printf("  Y=100,Z=50\n");
    printf("  X=+100,Y=-100\n");
    printf("  ?   (reports position)");
    printf("End each command with a newline.\n");
    printf("Begin typing motion commands\n");
    while(!feof(stdin)) {
        c = getc(stdin);
        parse_one_char(c);
        move_all();
    }

    if(power_down_on_exit) {
        word=flipbits;
        /* output low byte to data register */
        verbose_outb( (word & 0x00FF), base+0);
        /* output high byte to control register */
        verbose_outb( (word >> 8), base+2);
    }
}
#endif

```

31.7.2 编译驱动程序

程序清单 31.18 显示了使用 make 程序编译这个驱动程序所用的 Makefile。

程序清单 31.18 makefile

```

default: all

all: stepper.o stepuser

stepper.o: stepper.c ring.h stepper.h
    gcc -O -DMODULE -D__KERNEL__ -o stepperout.o -c stepper.c
    ld -r -o stepper.o stepperout.o ring.o

stepuser: stepper.c
    gcc -g -O -o stepuser stepper.c

ring.o: ring.c ring.h
    gcc -O -DMODULE -D__KERNEL__ -c ring.c

```

31.7.3 使用内核驱动程序

程序清单 31.19 中简单的一系列 shell 命令将说明这个驱动程序的使用法。

程序清单 31.19 驱动程序的使用法

```

# make the devices (once)
mknod /dev/stepper c 31 0
mknod /dev/stepper_ioctl c 31 1

#load the driver
sync; insmod ./stepper.o port=0x378

#give it something to do
cat "X=100,Y=100,Z=100" >/dev/stepper
cat "X=300,Y=300,Z=300" >/dev/stepper
cat "X=0,Y=0,Z=0" >/dev/stepper

#unload driver
sync; rmmod stepper.o

```

可以通过设置几个变量来修改驱动程序的运转。在用户模式里，变量 `delay` 设置移动之间睡眠的毫秒数；如果设置 `fastdelay`，则使用一个指定迭代数目的延迟循环。在内核模式中，可以通过跳过两次移动之间的指定数目的定时器报时使用 `skipticks` 来减慢移动。变量 `debug` 设置打印调试信息的数量。变量 `verbose_io`，如果非零的话，将导致在每次调用 `outb` 时打印调试消息。变量 `verbose_move`，如果非零，将导致在每次移动电机时打印调试消息。如果在内核模式驱动程序中设置 `verbose_io` 或 `verbose_move`，则要使用 `skipticks` 的值来降低调用 `printk` 的频率。变量 `base` 设定使用的并口接口的 I/O 端口地址，值 `0x3BC`、`0x378` 和 `0x278` 是最常见的。如果 `power_down_on_exit` 被设为非零值，将在退出用户模式程序或删除内核模式模块时关闭电机。变量 `do_io`，如果被设为零，将禁止 `outb` 调用，从而允许进行没有空闲并口的试验（在 `verbose_io` 和/或 `verbose_move` 设置的情况下）。这些变量中的一部分也可以用 `ioctl` 调用设定。

注意： 如果一个驱动程序在发出大量的 `printk`，你可能需要盲输入一个 `rmmod stepper` 命令，因为内核消息会使 shell 提示和输入回显滚动出屏幕。

注意，并口一定不能被其他设备驱动程序（如 lp0）使用。你可能需要卸载打印机模块或用 `reserve=0x378,4` 选项来引导系统。如果你试图使用中断并遇到问题，检查一下和其他设备的冲突（打印机卡上的中断通常不被打印机使用）并检查你的 BIOS，确保它没有被分配给 PCI 总线而被分配给了 ISA 总线或主板。

31.7.4 未来发展方向

我可能在将来对这个驱动程序进行一些改进，主要是对角度的三维移动的适当处理，及多个进程同时控制驱动程序的能力。还可能实现自动启动和 ioctl 驱动的启动和停止以便只在需要的时候使用中断、与其他进程同步及紧急停止。添加对 Linux 实时时钟扩展（需要定制的内核）的支持将允许使用快于每秒 100 脉冲的操作，但是，在维护这个标准字符设备接口时使用扩展则不是一件容易的事。添加一个 `stepper_select` 函数，并修改 `stepper_read` 和 `stepper_write` 函数，使它们返回比传递的计数更小的计数（count）而不是睡眠，将允许使用 `select` 和非阻塞 I/O。

31.8 其他信息资源

在因特网上有大量关于设备驱动程序的信息源，市场上也有全面介绍这一主题的书籍。

- Linux Kernel Module Programming Guide (Linux 内核模块编程指南)

<http://www.linuxdoc.org/LDP/lkmpg/mpg.html>

- The Linux Parallel Port Home Page (Linux 并行端口主页)

<http://www.torque.net/linux-pp.html>

- The PC's Parallel Port (PC 的并行端口)

<http://www.lvr.com/parport.htm>

- Linux Device Drivers^①

Alessandro Rubini 著, O'Reilly and Associates 出版, 1997 ISBN 1-56592-292-1 448pp

你可能会发现这些内核资源非常有用。其中提及的例程的头文件和源代码文件都是有用的参考资料。内核包含的大量现有的设备驱动程序也可以作为例子。

^① 译者注：这本书堪称 Linux 设备驱动程序设计的经典，目前已出了第 2 版。

31.9 小 结

写设备驱动程序是在 Linux 环境中最复杂的编程任务之一。它需要和硬件打交道，它很容易使系统崩溃，而且很难调试。没有你需要使用的各种内核函数的手册页，但是有更多的在线及印刷品可用。

从厂商处取得低层编程信息通常很难；你可能发现有必要对硬件实施逆向工程或从有良好声誉的厂商手中选择硬件。许多硬件有使编写驱动程序复杂化的严重设计缺陷或特性。但是，成功实现一个驱动程序是有益的，同时要提供必须的设备支持。

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □
书名 □ □ □ □ □ □ □ □ □ □ 编程指南（第二版） □
作者 □
页数 □ □ □ □ □
□ □ 号 □ □
出版日期 □